Tuscany SCA IN ACTION

Simon Laws
Mark Combellack
Raymond Feng
Haleh Mahbod
Simon Nash

MANNING

Tuscany SCA in Action

Tuscany SCA in Action

SIMON LAWS
MARK COMBELLACK
RAYMOND FENG
HALEH MAHBOD
SIMON NASH

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
180 Broad Street
Suite 1323
Stamford, CT 06901
Email: orders@manning.com

Manning Publications Co.
180 Broad Street
Suite 1323
Stamford, CT 06901

Development editor: Jeff Bleiel
Copyeditor: Linda Recktenwald
Typesetter: Marija Tudor
Cover designer: Marija Tudor

ISBN: 9781933988894
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 17 16 15 14 13 12 11

brief contents

contents

preface

What brought the five of us together to write a book on Apache Tuscany and Service Component Architecture (SCA)? We all had practical experience of how difficult and costly integration of applications and technologies can be, and we were excited about how Tuscany and SCA can help solve these problems. Having been involved with Tuscany and the SCA specifications from the early days, we understood the potential of this new technology and wanted to share it with you.

Although we'd been entertaining the idea of writing a book for a while, the event that made it possible was the completion of the SCA 1.0 implementation in Tuscany. With that we had a real implementation of a service-oriented infrastructure that we could use to explain SCA through examples. As well as implementing the SCA 1.0 specifications, Tuscany handles integration with many underlying technologies and enables users to focus on developing business solutions instead of worrying about infrastructure details. By writing this book we wanted to help our readers take advantage of the power of SCA and leverage the many technology choices that Tuscany offers.

Of course, we didn't see a point in repeating what the SCA specifications already provide. The specifications define SCA but don't explain how to use it. Rather, we chose to address how to use SCA with Tuscany by showing working examples and sharing best practices. There are articles available that give a high-level overview of Tuscany and SCA, and there's some detailed technical information on the Tuscany website, which assumes a good understanding of the technology. What's been missing until now is a hands-on introduction and guide that explains the capabilities of

Tuscany and SCA and shows by practical examples why these are useful to application developers. This book provides that "missing link."

We hope that by reading this book you come to share our enthusiasm for SCA as a rich programming model that makes it easy to create flexible service-based applications, and that you discover how Tuscany's wide range of technology support can help you overcome the challenges posed in integrating service-based applications.

acknowledgments

Writing any book presents a challenge, and this is particularly true for a geographically distributed group of technologists. You're able to hold the finished book now because of significant help that we received from other people.

First of all, we'd like to thank the Tuscany community for creating and using the software, getting involved on the Tuscany mailing lists, and motivating us to write this book in the first place. We hope this book helps to add some detail to the many topics that we've talked about on the mailing lists over the last few years.

The process of writing has been a voyage of discovery for all of us, and the shape and focus of the book wouldn't be what it is without the many reviewers who've given their time to read and comment on the manuscript during its development. This includes Jeff Davis, Mykel Alvis, Ara Ebrahimi, Doug Warren, Alberto Lagna, Jeff Anderson, Mike Edwards, Kevin Williams, Marco Ughetti, Robert Hanson, and Tray Scates. We're also grateful to the MEAP subscribers who've given us valuable feedback via the Manning forum.

The team at Manning has been particularly helpful and understanding. Megan Yockey originally commissioned the book, and Marjan Bace gave us the confidence to get started. Most important, Jeff Bleiel has been our constant companion during the development of the book, nudging us in the right direction and giving us unfaltering encouragement. We also thank the many other members of the Manning team who helped us behind the scenes, including production team members Mary Piergies, Linda Recktenwald, Allison Cichosz, Katie Tennant, and Janet Vail, as well as Doug Warren, who did a final technical review of the manuscript during production.

It goes without saying that this exercise has taken up many weekends and evenings, and so our final thanks go to our families.

SIMON L To Maddy, thank you for sticking with it. You can have me back now.

MARK I would like to thank my loving and beautiful wife, Amy, for her enduring patience and support throughout this project. Perhaps I shall spend less time at my computer now that it's finished. Thanks to my daughters, Emily (age 3) and Chloe (age 1), for their understanding that Daddy sometimes needed to write his book rather than play princesses or read nursery rhymes. Thanks to our family and friends for their support and for keeping my girls company while I worked on this book.

RAYMOND I greatly thank my wife, Tao, for accepting my endless excuses to spend hours of weekend time writing the book. Thanks to my sons, Thomas and Jerry; you can now have more time with me and be proud of your Daddy for publishing a book.

HALEH I thank my dear family—Bahman, Aurash, and Armaan—who always support and encourage my endless projects.

SIMON N To my wife, Charlotte, thank you for your understanding and patience as I spent many hours at the computer and on the phone doing "book stuff," including a considerable amount of time while we were traveling in New Zealand. To my sons, David and Adam, and my parents, thank you for your interest and encouragement in this venture.

about this book

The Apache Tuscany open source project was created to overcome the challenges associated with creating, deploying, and managing service-based applications—in particular, applications made of many components, potentially written using different programming languages, using different data formats, and communicating with various communication protocols. Apache Tuscany, or just *Tuscany* for short, provides the infrastructure that solves this problem and allows companies to focus on developing business components rather than worrying about managing and maintaining the underlying infrastructure.

The Tuscany project encompasses a number of different technologies, but the glue that binds everything together is provided by the Service Component Architecture (SCA). *Tuscany SCA in Action* focuses on SCA and explains how composite applications can be developed easily using the Apache Tuscany SCA Java runtime, or *Tuscany SCA* for short. This book is a tool for learning SCA and Tuscany. It provides detailed practical examples and is a guide for those wanting to learn how to create real applications. The source code for the examples in this book is available from the Apache Tuscany project at http://tuscany.apache.org/sca-java-travel-sample-1x-releases.html, or from the publisher's website at www.manning.com/TuscanySCAinAction.

How the book is organized

Tuscany SCA in Action is divided into four parts, plus four appendixes. The first part introduces SCA as a programming model and Apache Tuscany as the platform for developing applications using SCA. In this part of the book we introduce the TuscanySCATours travel-booking application. The travel-booking application is developed in

the book as we cover various aspects of Tuscany and SCA. Part 2 looks in more detail at SCA's support for developing services and assembling them into composite applications. It starts by explaining the SCA-supported component interaction patterns. It continues with a detailed description of the various implementation, binding, databinding, and policy technologies that Apache Tuscany supports. Part 3 explores techniques for deploying the travel-booking application locally or into a distributed environment. Apache Tuscany supports a number of technologies out of the box to facilitate integration with a variety of existing technologies. In addition, it offers an extensible architecture that allows users to extend it with new technologies. Part 4 explains the Apache Tuscany architecture and how it can be extended to support new technologies.

After reading this book you'll have a thorough understanding of SCA and its benefits for your business. You'll learn this through practical examples that are available as runnable applications from the Apache Tuscany website. You'll also learn how to join the community of users and developers who work with Apache Tuscany and extend Apache Tuscany to support new technologies. Let's look at how each part is divided into chapters.

Roadmap

Part 1 consists of chapters 1 through 3. Chapter 1 explains what Apache Tuscany is and highlights its benefits. It also introduces SCA, including a quick jumpstart for creating an application using SCA and Tuscany. Chapters 2 and 3 demonstrate most of the features of SCA at a high level using examples. They highlight how SCA can be used to assemble components into applications when the components may have been developed with a variety of technologies, using a variety of data formats and communication protocols.

Part 2 of the book focuses on understanding the detailed features of Apache Tuscany. This part consists of chapters 4 through 10. Chapter 4 covers interaction patterns in composite applications. Chapters 5 and 6 provide examples of developing components and services using Java, BPEL, Spring, and scripting technologies. Chapter 7 describes how components can be assembled and easily reassembled, using SCA bindings for a variety of technologies including Web Services, RMI, and more. In chapter 8 we focus on the client side and how Web 2.0 can be used with SCA to provide a flexible web client.

Chapter 9 explains how SCA services can use different data formats to interact with one another. It covers Service Data Objects (SDO) and Java Architecture for XML Binding (JAXB) as examples.

By now we've covered how to create and deploy composite applications. In chapter 10 we talk about how to apply quality of service features to these applications using policies, for example, to handle security configuration.

Now that we've developed a flexible application and shown how to deploy it, in part 3 we look at the choice of host platforms that Tuscany supports. Tuscany can be

embedded into a variety of host platforms, for example, Apache Tomcat, Jetty, and Apache Geronimo. Chapter 11 covers hosting environment choices. Chapter 12 finishes off part 3 by describing how the pieces discussed in the earlier chapters come together to complete the travel application.

Part 4 of the book is for developers who'd like to learn to extend Apache Tuscany to support new technologies. This part consists of chapters 13 and 14. Chapter 13 describes the architecture of the Tuscany SCA Java runtime. Chapter 14 talks about how Tuscany can be extended to add a new component implementation and a new binding type. It also discusses how Tuscany seamlessly handles protocol format differences between components through its databinding framework and how that too can be extended.

Tuscany SCA in Action has four appendixes. Appendix A helps you set up your environment to run the examples in the book. Appendix B shares some thoughts on likely future directions for the Apache Tuscany project. Appendixes C and D include copies of the OSOA SCA specification license and ASF2 license, respectively.

Who should read this book

Tuscany SCA in Action is for all enterprise developers who care about creating reusable services and assembling those services into flexible composite applications (business applications). The particular focus is on freedom of choice of technology for developing component implementations, using communication protocols, and handling data formats. The book guides you through learning SCA and Tuscany using code examples and concludes with the assembly and deployment of the travel-booking application.

Although a major portion of the book is focused on developing applications using Tuscany and SCA, part 4 talks about how to extend Tuscany to support new technologies. This part is particularly relevant for architects and developers who would like to extend Tuscany to embrace other technologies not currently supported by Tuscany and to learn about how to get involved with the Tuscany open source project.

The scope of the Tuscany project is quite broad, and so we assume that you're familiar with some of the basic techniques and technologies on which the Tuscany project builds, in particular the following:

- The Java programming language
- XML, XML Schema, and the use of XML namespaces
- Web Services Definition Language (WSDL) and the use of tools for transformation between WSDL and Java classes

If you need more information on any of these subjects, then the internet is your friend. Many online resources are available that you can refer to.

Code conventions

The book contains many code examples. These examples will always appear in a `fixed-width code font`. Any class name, method name, or XML fragment within

the normal text of the book will appear in `code font` as well. All runnable code pieces will appear as listings. Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered bullets link to explanations that follow the listing.

Author Online

Purchase of *Tuscany SCA in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/TuscanySCAinAction. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct in the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the authors can take place. It isn't a commitment to any specific amount of participation on the part of the authors, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

SIMON LAWS is a committer for the Apache Tuscany project focused on building the Java runtime for the Service Component Architecture. He's been working in commercial software development for 23 years and has a general interest in distributed component-based technology. In the past he's worked on both PHP and C++ runtimes for SCA. He's a member of the IBM Open Source SOA project team and lives and works in Hampshire in the UK.

MARK COMBELLACK graduated with a degree in computer science in 1994 and has been using the Java programming language since JDK 1.0.1 in 1996. Mark is currently working as a software developer specializing in the development of Java application servers for the telecommunication industry. Previously, he has developed software for interactive television and video on demand as well as other telecommunication projects for British Telecommunications. He is a committer and a member of the Project Management Committee of the Apache Tuscany project. For two years, he was co-chair of the SCA-J OASIS Open CSA technical committee standardizing SCA in the Java programming language.

RAYMOND FENG is a PMC member and committer of the Apache Tuscany open source project. He has been actively contributing to Tuscany to build the Service Component Architecture runtime for more than 4 years. His expertise spans most of the areas in the project, including core architecture, Java EE, OSGi, Web Services, XML, and data-bindings. Prior to this role he was a developer and team lead for the IBM WebSphere Process Server products, where SCA was originally invented and implemented. Raymond has been a pioneer and veteran in SCA runtime development since 2002. He

also contributed to the SCA and Enterprise OSGi specifications as a member of OASIS Open CSA committees and OSGi Alliance. Raymond has spoken in many conferences to evangelize SCA, including JavaOne, ApacheCon, and SOAWorld.

HALEH MAHBOD directed the development of SCA from its inception at IBM; SCA was then contributed to Apache. As one of the founding members of the Apache Tuscany project, she has made significant contributions to creating Tuscany's open source community by promoting the technology and Tuscany worldwide. Haleh has a BS degree in computer science from U.C. Berkeley. With over 20 years of experience, Haleh has held various leadership positions as architect and director in different startup and Fortune 500 companies. She has been at the forefront of a number of innovative technologies and products with particular focus on data and application integration. Haleh's varied experience and extensive work with enterprise-level customers struggling with software integration cost and complexities has been a catalyst for her to innovate with technologies such as SCA to address these challenges. In her spare time Haleh is an accomplished photographer and philanthropist.

SIMON NASH is a Tuscany PMC member and committer and has made significant contributions to the OASIS specifications for the SCA 1.1 standard. Simon started developing software in 1970, and he has been at the forefront of a number of innovative technologies with particular interests in programming languages, communication technology, parallel processing, object technology, and simplifying software development. Simon was IBM's CTO for Java technology from 2001 to 2003, and he was an IBM Distinguished Engineer from 2003 until he retired from IBM in 2008. His other notable career achievements include Object REXX (1988–1993) and RMI-IIOP (1997–2001).

about the title

By combining introductions, overviews, and how-to examples, the *In Action* books are designed to help learning and remembering. According to research in cognitive science, the things people remember are things they discover during self-motivated exploration.

Although no one at Manning is a cognitive scientist, we are convinced that for learning to become permanent it must pass through stages of exploration, play, and, interestingly, retelling of what is being learned. People understand and remember new things, which is to say they master them, only after actively exploring them. Humans learn in action. An essential part of an *In Action* book is that it's example driven. It encourages the reader to try things out, to play with new code, and explore new ideas.

There is another, more mundane, reason for the title of this book: our readers are busy. They use books to do a job or solve a problem. They need books that allow them to jump in and jump out easily and learn just what they want, just when they want it. They need books that aid them *in action*. The books in this series are designed for such readers.

about the cover illustration

The figure on the cover of *Tuscany SCA in Action* is captioned "Berger Des Garrigues," indicating a shepherd from the southern regions of France around the Mediterranean Basin. *Garrigue* is a type of low, soft-leaved scrubland typically found in Mediterranean forests and woodlands in France and Spain, and often composed of kermes oak, lavender, and thyme. The illustration is taken from a nineteenth-century edition of Sylvain Maréchal's four-volume compendium of regional dress customs published in France. Each illustration is finely drawn and colored by hand. The rich variety of Maréchal's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns or regions. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Maréchal's pictures.

Part 1

Understanding Tuscany and SCA

Tuscany SCA In Action teaches you to use the Service Component Architecture (SCA) through practical examples developed using the Java SCA runtime from the Apache Tuscany open source project. This part of the book provides a broad introduction to SCA and Apache Tuscany using a travel-booking scenario. It presents an overview of Apache Tuscany and helps you to set up your environment so that you can develop and run the examples in this book. This is all covered in three chapters.

In chapter 1, "Introducing Tuscany and SCA," you'll learn about the Apache Tuscany open source project and the travel-booking scenario and be introduced to SCA at a high level. By the end of this chapter, you'll have an appreciation of SCA and will be able to run a small part of our travel-booking application.

Chapter 2, "Using SCA components," covers SCA in greater detail. You'll gain a better understanding of SCA and use it to create more of the components that compose the travel-booking application.

Chapter 3, "SCA composite applications," covers the deployment of SCA-based applications. Here, you'll learn about the flexible deployment model that Tuscany SCA offers and experiment with deploying the travel application in a local or distributed environment.

By the end of part 1, you'll know about Tuscany and SCA and will have built and deployed a simple travel application.

1 Introducing Tuscany and SCA

This chapter covers

- Exploring SCA and Tuscany
- Learning basic SCA concepts
- Developing your first SCA application

Businesses are always looking for ways to lower the cost of creating and maintaining business applications. One popular approach to business application development, often called Service Oriented Architecture (SOA), is to adopt a model where business functions are described as well-defined services that can be used to compose working applications.

SOA is an attractive idea, but putting it into practice can be difficult. Business computing environments typically contain many different technologies, and the integration of these technologies can be complex. In a single application you can be joining Java objects, Business Process Execution Language (BPEL) processes, browser-based clients, and Ruby scripts using web services, as well as Java Message Service (JMS) and JSON-RPC protocols, to name but a few.

What's needed is a common way to describe an assembly of distributed services regardless of the technology used to implement and connect them. Step forward the Service Component Architecture (SCA) and the Apache Tuscany project.

Apache Tuscany is an open source project developed by the Apache Software Foundation. The Tuscany software is freely available from the project website (http://tuscany.apache.org) under the Apache 2.0 License. The software is a lightweight infrastructure that implements Service Component Architecture (SCA), Service Data Objects (SDO), and Data Access Service (DAS) technologies and provides seamless integration with many other technologies. This book is about Tuscany's Java implementation of SCA, which is what we mean when we use the term *Tuscany SCA*.

The SCA specifications are the foundation upon which Tuscany SCA is built. The first version of SCA specifications (v1.0) was developed by a consortium of companies called the Open Service Oriented Architecture (OSOA) collaboration. The specifications are published via the collaboration's website (http://www.osoa.org). These are the specifications that the Tuscany Java SCA v1.x runtime and this book use.

OSOA and OASIS versions of the SCA specifications

Following the release of the SCA v1.0 specifications from OSOA, the SCA specifications were donated to the OASIS Open Composite Services Architecture (CSA) Member Section (http://www.oasis-opencsa.org/). Work is ongoing at OASIS to standardize v1.1 of the SCA specifications. The Tuscany 1.x runtime and this book are based on the completed v1.0 specifications from OSOA. Appendix B covers the direction that Tuscany 2.x is taking beyond what's available in Tuscany 1.x. The fundamentals of what you learn about SCA in this book apply to both versions.

SCA provides a technology-neutral assembly capability for composing applications from business services. The services themselves can be developed and connected using many different technologies. If you look at the Tuscany project website, you'll find subprojects providing Java language and C++ (also known as native) implementations of SCA. You'll also find Java language and native implementations of SDO and DAS, which provide ways of handling and persisting data. SDO and DAS aren't prerequisites for using SCA. Although this book concentrates on Tuscany's Java SCA runtime, in chapter 9 we do use SDO when building SCA service interfaces. If you want to know more about SDO, DAS, or the native runtimes, then the Apache Tuscany website (http://tuscany.apache.org) is a good place to start.

We'll start this chapter by taking our first high-level look at SCA and Tuscany. Then we'll look at how an example travel-booking application can be described using SCA. This exercise sets the scene for building and running your first SCA application in section 1.3.

There are already many SOA-related technology choices out there. In the last section of this chapter we look at how Tuscany and SCA are able to integrate with and complement other popular SOA technologies.

The samples used in this chapter, and in the rest of this book, can be downloaded following the instructions in appendix A. The source code for the samples is available in the Tuscany project, and the samples are accompanied by a README file that describes the structure and the operation of the samples.

In this chapter you'll gain a high-level understanding of the Tuscany software and the advantages of SCA, and you'll build your first composite application. This will get you ready to dive into the rest of the book and explore what else Tuscany SCA has to offer.

1.1 The big picture

SCA uses a range of terms, some of which will sound familiar and others that are new. It's important to appreciate what SCA means when it talks about such things as components, services, references, and composites. These terms will be used repeatedly throughout the book, so we'll start here by giving a high-level introduction of what it means to assemble applications from SCA components and what the various parts of the resulting assembly are called.

Assembly is at the core of SCA, so much so that the central SCA specification concentrates on defining what's called the Assembly Model. The SCA Assembly Model defines an XML language for assembling components into applications and provides the framework into which extensions are plugged to support the wide variety of implementation and communication technologies that are available today. In the next sections, we'll first provide an overview of the SCA Assembly Model and then give a quick summary of how Tuscany is architected to support SCA. This will provide sufficient background for understanding the details throughout the rest of the book.

1.1.1 The basics of SCA

SOA promotes the benefits of constructing large and complex enterprise systems out of well-defined and sometimes replaceable component parts called *component services*. SCA describes an Assembly Model for doing just that.

An SCA service provides a reusable piece of business function and has a well-defined interface that identifies how it can be called to provide that function. An application broken down into a set of well-defined services significantly reduces the complexity of development as well as its long-term maintenance by isolating change and simplifying testing. The challenge then becomes how to assemble the cooperating network of services to provide maximum flexibility and reuse while maintaining the integrity of each service. Figure 1.1 shows a web shopping application that uses a set of connected services to allow the user to browse a catalog, add items to a shopping cart, and then pay for the items at checkout time.

Figure 1.1 demonstrates that the web shopping example is made up of services that are developed in various technologies and communicate using different protocols. This mix of technologies is typical of today's applications.

The danger with the usual approach to application development is that technology integration logic can often become intermingled with business logic. For example, we may call remote web services by using a web service provider API directly from business

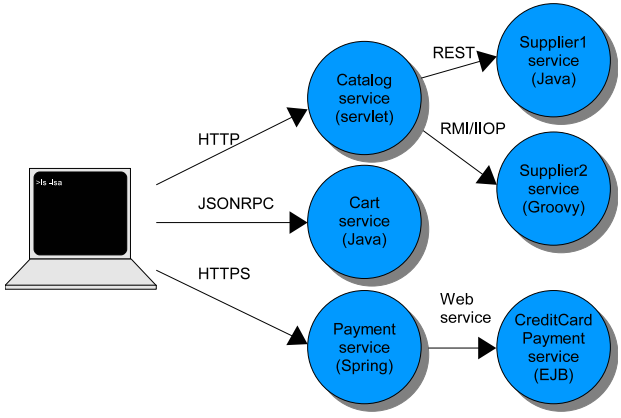


Figure 1.1 A web shopping application built from cooperating services showing the typical variety of technologies use to implement and connect services

logic. This makes services hard to build, hard to maintain, and hard to deploy and reuse. A higher level of abstraction is required to describe the assembly of such services.

The Service Component Architecture has been designed to address this issue. It does this by defining an Assembly Model that provides a clear separation between business logic and other infrastructure concerns.

Figure 1.2 shows the main artifacts of the SCA Assembly Model by taking the Payment and CreditCardPayment functions from figure 1.1 and mapping them to SCA components and services. We'll use a style of diagram that's similar to those the SCA specifications use to show composite applications, but we'll extended it to show bindings. In an SCA application the *component* is the basic building block. A collection of components that make up all, or part of, an application is called a *composite* and is described using simple XML constructs. Figure 1.2 gives an overview of a composite

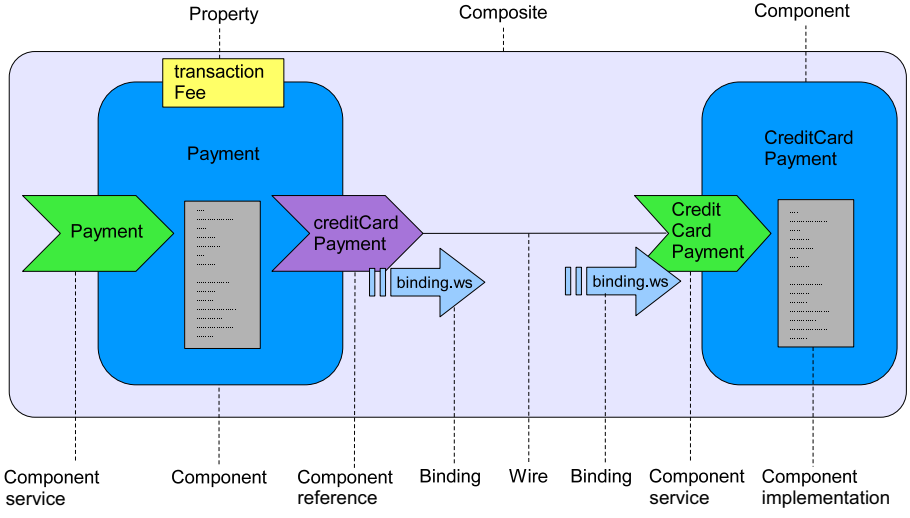


Figure 1.2 The Payment and CreditCardPayment components from the web shopping application presented as SCA components in order to show the main artifacts of the SCA Assembly Model

application with two components, CreditCardPayment and Payment, that are wired together using the web service binding.

A component is a configured instance of some business logic. It provides services and can use services. Every SCA *service* has a name and an interface. The interface defines the operations that the service provides. You might be more familiar with other terms in place of *operation*, such as *method*, *function*, or *request*. A component can provide one or more services. The business logic of a service is provided by a component's *implementation*, for example, a Java class containing business logic for CreditCardPayment processing.

A component implementation can be configured by defining *properties*. In our example, transactionFee is a property for the Payment component. A property value is set through configuration and is made available to the component implementation in a way appropriate to the implementation language in use.

SCA components call a service using a *reference,* for example, the creditCardPayment Service Reference in the Payment component. The component implementation is given access to references in an implementation language–specific way.

The connection between the reference and the service is called a *wire*. References are wired to services, and so a network of connected components is described within a *composite* application.

At a high level that's all there is to it. Using these simple constructs, applications of arbitrary complexity can be composed using a concise, precise, and standardized component Assembly Model.

Components are implemented using a regular programming language of your choice, such as Java, Ruby, or BPEL, or frameworks such as Spring, Java EE, and OSGi. This is called a *component implementation*.

What's more, this assembly approach allows components implemented with one technology, say the Java language, to be connected to components implemented in another technology, say BPEL. The detail of a component's implementation is abstracted away from the other components that it's connected to.

Building on this idea of abstraction, the technology used to join components together is unrelated to how the components are implemented. This is what SCA calls a *binding*. Today we may choose to connect the Payment and CreditCardPayment components using web services. Tomorrow we may choose to exploit the asynchronous and assured delivery properties of a JMS provider to connect the components. We could even support both web services and JMS. We can do this by changing the configuration of the assembled application and without changing the component implementations.

The main point here is that the Assembly Model is at the heart of SCA and Tuscany. All the extensions that we'll describe in this book are based on this simple idea. The Assembly Model is compelling not only because of the flexibility it brings to your applications but also because of the flexibility it brings to the process of application development.

A good example of this is how long it takes to build a web service today. It probably takes no more than a few minutes with modern software tools to generate WSDL and client proxies. SCA brings this level of productivity to the problem of wiring up services regardless of the technology used.

This is particularly powerful when your application development takes an incremental and prototype-driven approach. The Tuscany community has worked hard in building the Java SCA runtime to make the tedious and fiddly things simple and automatic. For example, imagine that you want to build a service that will be available over web services and JMS at the same time. First, SCA makes this configuration easy to describe. Second, Tuscany makes this configuration easy to test, with no special configuration required to automatically start web service containers and JMS providers. Come deployment time, you can then adjust the configuration of the Tuscany runtime and use the container of your choice.

Chapter 2 takes a much more detailed look at SCA components. So now let's move on and take a quick look at how the Tuscany Java SCA runtime is structured.

1.1.2 Tuscany's Java runtime for SCA

It's useful to take a high-level look at the structure of the Tuscany Java SCA runtime at this point so that you can better understand the relevance of the various chapters in this book.

Tuscany Java SCA offers a lightweight runtime that can be used out of the box to build composite applications using SCA. Alternatively, the Tuscany libraries can be embedded in other applications so they too can host SCA composite applications.

The Tuscany runtime has a modular and pluggable architecture so users can choose the functionality that they need and discard the rest. It's easy to manage the software footprint to suit each individual requirement.

At a high level the Apache Tuscany SCA Java runtime can be divided into a core infrastructure and a set of extensions that extend the core to work with various technologies. Altogether this is referred to as the SCA runtime and is shown in figure 1.3. Let's look at each of these building blocks in turn.

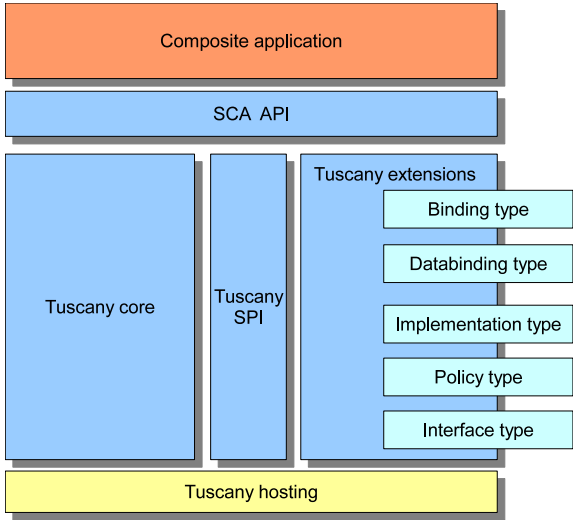


Figure 1.3 The main building blocks of the Tuscany SCA Java runtime

COMPOSITE APPLICATIONS

The composite application is shown in the top box in figure 1.3 and represents the business application we're building with SCA and Tuscany. It's described using the Assembly Model XML that the SCA specifications define. It defines wired components whose implementations reference the artifacts required to run the application, such as Java class files and BPEL process files. This topic is covered in chapter 3 in more detail.

SCA API

The SCA API sits between the composite application and the rest of the runtime in figure 1.3. It allows component implementations in the composite application to interact with the runtime. The SCA API is implementation language specific; for example, a Java Common Annotations and APIs specification describes the version of the SCA API for the Java language.

TUSCANY CORE

To the left of figure 1.3 is the core infrastructure. This supports construction of components and their services, the assembly of components into usable composite applications, and the management of the resulting applications. We discuss the Tuscany runtime architecture in chapter 13, which gives you an idea of how the Tuscany core operates.

TUSCANY EXTENSIONS

The Tuscany SCA runtime is designed to be extensible in order to accommodate the large range of existing technologies and to allow new technologies to be adopted as they're developed. The basic plug points are shown on the right-hand side of figure 1.3 and consist of binding, databinding, implementation type, policy, and interface.

Bindings provide support for different kinds of communication protocols, such as SOAP/HTTP web services, JSON-RPC, and RMI. Components use these to interact with one another. Chapter 7 describes how to use various binding extensions.

Databindings provide support for different data formats that can pass between services, such as SDO, JAXB, and AXIOM. The Tuscany core provides a databinding framework that enables services using different data formats to work seamlessly with one another. This frees the developer from defining explicit data format conversions. Chapter 9 talks in more detail about how data is represented and transformed.

The implementation type extension in figure 1.3 provides support for different programming languages and container models, such as the Java language, BPEL, and Spring, and scripting languages like Ruby. Tuscany users can develop or use services written with different languages in their composite applications. We've devoted chapter 5 to the SCA Java implementation type and chapter 6 to the Spring, BPEL, and scripting implementation types.

The policy extension in figure 1.3 separates infrastructure setup concerns from the development of services. This provides flexibility to adjust infrastructure-related policies such as security and transactions without impacting the code. Chapter 10 covers policy in more depth.

Finally, the interface extension allows service interfaces to be described using a variety of technologies. Currently, Java interfaces and WSDL are the two supported

means for defining service interfaces. Chapter 2 gives a good description of the role interfaces play.

TUSCANY SPI

Although Apache Tuscany supports many popular technologies in the form of extensions, new extensions can be added easily using the *Tuscany SPI*. Chapter 14 gives an introduction to building Tuscany extensions.

TUSCANY HOSTING

The Apache Tuscany SCA Java implementation has a modular architecture. This makes Tuscany more easily extensible and simplifies integration with other technologies. It allows Tuscany adopters to pick and choose modules that they're interested in exploiting.

In particular, the project's modular structure provides for a lightweight and flexible packaging and distribution mechanism. A set of Tuscany hosting modules allows developers to choose from a variety of options for how they want their composite application to run, for example, as a command-line application or within a web application. Tuscany already runs on a variety of hosting platforms, including Apache Tomcat, Jetty, and Apache Geronimo, and many commercial application containers, such as IBM WebSphere, and can easily be extended to include others. Chapter 11 describes the various hosting options that Tuscany offers.

Now that you know a little about SCA and Tuscany, let's try Tuscany out for real. In the next section we describe how a simple application can be composed from an assembly of components in preparation for building the application in section 1.3.

1.2 Designing a sample composite application

The strengths of SCA and Tuscany can best be demonstrated through scenarios and examples. Let's introduce an imaginary business called TuscanySCATours that's building a travel-booking application and is looking for an extensible architecture to accommodate its current needs and predicted future growth.

The travel-booking application is used throughout the book to demonstrate the various features of the Tuscany Java SCA runtime. Like many applications, our application starts small and needs to grow and change over time. You guessed it—using an SCA composite application is an ideal way to provide this kind of flexibility.

In this section we introduce you to the scenario and show how the application can be described using a composite application.

1.2.1 The travel-booking application

TuscanySCATours is a newly formed travel agency. Initially, the agency intends to offer a limited selection of canned travel packages for U.S. customers that include flight, hotel, and airport transfers. Depending on the initial success of the travel agency, TuscanySCATours plans to extend its offerings to include travel packages for customers from other countries, optional car rentals, and the ability to create customized travel packages.

The first version of the travel application is simple. The user uses predefined trip-booking codes to populate the shopping cart and purchase a trip. We assume that the web application displaying the travel package catalog, and which provides the predefined trip-booking codes, has been implemented using off-the-shelf software that doesn't use SCA. The browser-based SCA application allows the user to take the codes of the selected trips and add them to the shopping cart.

TuscanySCATours is using SCA to implement its trip-booking and payment systems with two components named TripBooking and ShoppingCart. TuscanySCATours doesn't organize trips itself but buys them from a partner called GoodValueTrips.

GoodValueTrips uses software that wasn't originally developed using SCA, and so a third component called TripProvider wraps this existing non-SCA code.

For credit card payment processing, TuscanySCATours communicates with an existing software package running outside the Tuscany Java SCA runtime.

The diagram in figure 1.4 shows this travel-booking application. It's a high-level architectural overview that shows components as simple boxes with no details of what's inside them, with one important exception: where one SCA component makes use of a service provided by another SCA component, the diagram shows these interactions as solid arrows. The dashed boxes and dashed arrows represent non-SCA software packages and their own interactions.

Even with this simple scenario, this looks a little complicated for a first application. But SCA is well suited to dealing with this combination of existing and new software, and this is exactly the kind of scenario you're likely to face when approaching an SOA project.

Now that you understand the basic architecture of the application, the next step is to translate this high-level block diagram into SCA components and services. In the next section we show how the boxes and arrows in figure 1.4 are represented using SCA.

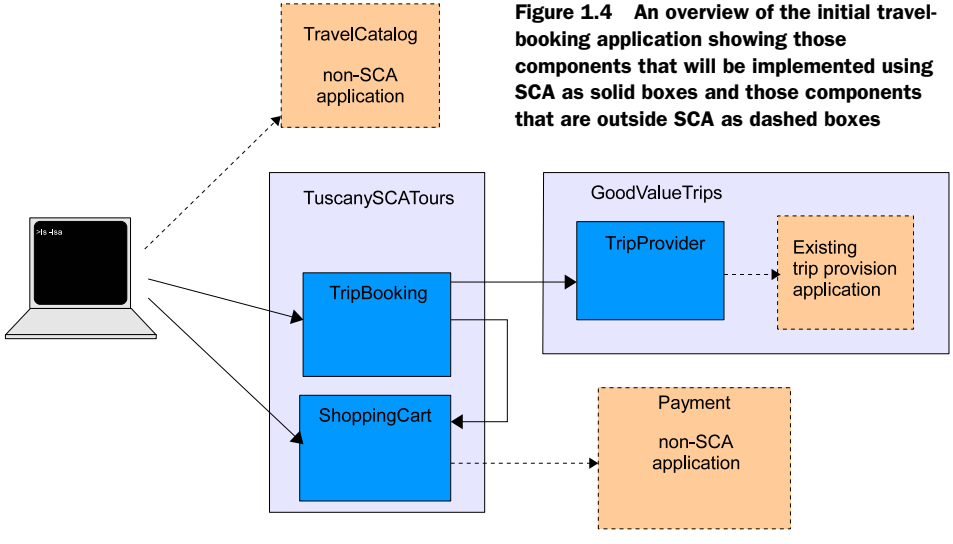


Figure 1.4 An overview of the initial travel-booking application showing those components that will be implemented using SCA as solid boxes and those components that are outside SCA as dashed boxes

1.2.2 SCA components, services, and references

The solid arrows in figure 1.4 represent service interactions between components, with the arrowhead attached to the component providing the service. Let's add more detail to our architectural overview by including the services and references that the components provide. For example, in figure 1.5, the TripBooking component has a single service named Bookings and two references named `mytrips` and `cart`.

Wires connecting references to services are shown as plain lines without arrowheads. For example, in figure 1.5, the `mytrips` reference of TripBooking is wired to the Trips service of TripProvider. No arrowhead is needed because the direction of a wire is always from its reference to its target service.

To validate the design of the travel application using SCA components and services, it's useful to walk through the end-to-end message flows and through the various components and services involved. In the next section we'll describe a user scenario and show how this translates into SCA service interactions.

1.2.3 A user scenario demonstrating service interactions

To illustrate how the services in figure 1.5 work and interact, we'll use a simple scenario of a customer, called Mary, making a travel booking. Stepping through a user scenario is an important part of validating the software design because it exposes any problems with the chosen structure of components and services and allows corrections to be made before incurring the expense of creating an implementation. To keep the scenario as simple as possible, we won't include the transactional coordination aspects that can be associated with a travel-booking scenario. The following steps refer to the numbers in figure 1.5:

1. *Selecting a trip*—Mary wants to visit Italy. She browses the TuscanySCATours website, looking at the various packages that are available. She decides she'd like to

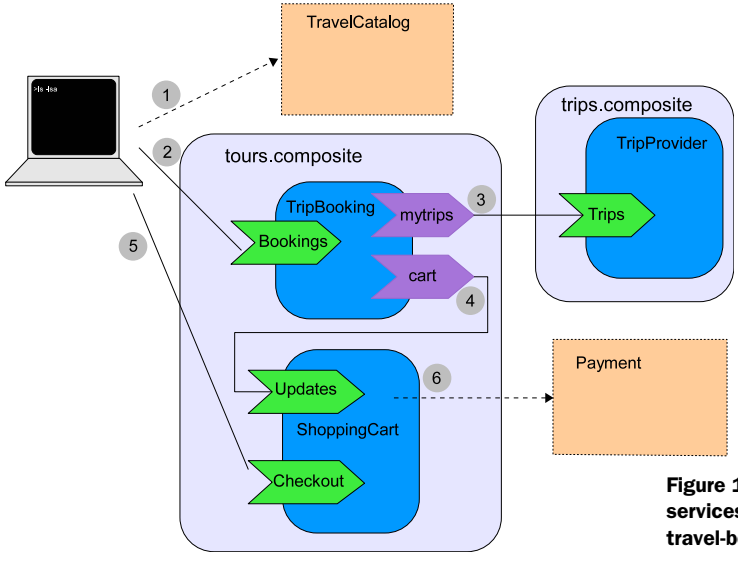


Figure 1.5 The components, services, and references of the travel-booking application

book the Florence and Siena trip, departing on April 4. The booking code for this trip is FS1APR4.

2. *Booking a trip*—To reserve a place on this trip, Mary's browser-based client software sends a newBooking request to the Bookings service of the TripBooking component. The parameters for this request are the trip-booking code FS1APR4 and the number of people traveling, in this case one.
3. *Checking that the trip is available*—The newBooking request is received by the Bookings service of the TripBooking component. This service needs to find out whether the requested trip is available. It does this by calling the checkAvailability operation that the Trips service of the TripProvider component provides. There are enough unsold places within the trip to satisfy Mary's request, so the Trips service indicates this by returning a reservation code to the Bookings service.
4. *Adding the trip to the shopping cart*—The Bookings service uses the Updates service of the ShoppingCart component to add the selected trip to Mary's shopping cart and then responds to the browser client, confirming Mary's reservation for the trip she requested.
5. *Checking out*—Now that Mary's booking is confirmed, she needs to pay for it. For this, Mary's client software uses the Checkout service of the ShoppingCart component. This service has a `makePayment` operation that Mary's client software uses to send her credit card details.
6. *Processing the payment*—The `makePayment` operation uses a third-party credit card processing service to validate Mary's credit card details and ensure that she has enough funds to make the payment. Everything is fine, so the `makePayment` operation returns to Mary's client software, confirming that the payment was accepted. In the event of a problem with the payment, the `makePayment` operation would throw an exception back to the client, with the exception type and exception data giving details of the payment problem.

Notice that everything needed to make and confirm the booking was done by invoking services. Some of these services are using other services as part of their processing. For example, the newBooking operation of the Bookings service used the checkAvailability operation of the TripProvider service, and the `makePayment` operation of the Checkout service invoked a third-party credit card processing service. Using existing services as part of the implementation of a new service is called composition, and the result is a composite application. Making the creation and deployment of composite applications easy to do is one of the objectives of SCA and Tuscany.

An SCA composite application describes the way that component services are wired together. This description may be explicit about the physical location of component services. Alternatively, it may omit this information and defer to the Tuscany runtime to determine the physical location of deployed component services. Whichever approach is taken the composite application will still describe how component services are logically wired to form the application.

You've seen how the architecture and design of a business application can be expressed using the basic elements of the SCA programming model: components, services, and references. You've also seen how SCA services and non-SCA services can be combined within a business application. Let's get our hands dirty and implement some of the parts of this simple scenario.

1.3 Implementing a composite application

In the previous section we looked at the architecture and design of the travel-booking application. In this section, we'll build the components of the application. We'll do this in two stages. First, we'll cut straight to the action and build a single component and run it to see how it works. Second, we'll take a more studied look at how to wire components together into a running application.

Let's start by building and running the TripProvider component. You'll find the launcher for this sample in the sample directory launchers/jumpstart. A launcher is a simple Java program that loads and runs the sample.

We've chosen the TripProvider component because it's simple. It provides a single service and doesn't use any references. Once we have this first component running, we'll build and wire the other two components.

1.3.1 A jump-start to building and running your first SCA component

For this first example we'll create a simple version of the GoodValueTrips company's TripProvider component. There are no particular restrictions to the environment you can use to build this application. Appendix A gives an overview of how to use Tuscany with Ant, Maven, and Eclipse, and the sample code comes with pom.xml and build.xml files for Maven and Ant users. All of the following code is provided with the book samples in the contribution/introducing-trips and launcher/jumpstart directories, so you won't need to type it in manually.

Our goal here is to send a test message to the TripProvider component from a Java program to demonstrate that it works. Figure 1.6 shows the TripProvider component as we'll build it here. The Trips service is shown as the right-facing arrow to the left of the TripProvider component box.

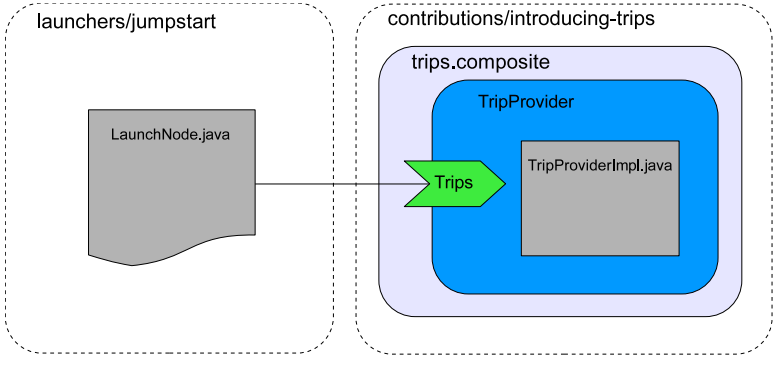


Figure 1.6 The configuration of the first GoodValueTrips TripProvider component we'll build

We'll follow these five steps to create the TripProvider component:

1. Design the Trips.java service interface.
2. Build the TripProvider.java component implementation.
3. Build the XML trips.composite file to define the SCA TripProvider component.
4. Package TripProvider.java and trips.composite into the scatours-contribution-introducing-trips.jar contribution. Let's for now assume that a contribution is a package that contains the code that you want to run and its related artifacts. This concept is explained in detail in chapter 3, but this explanation is sufficient for now.
5. Create a simple launcher to load the contribution and test the Trips service.

Sample contribution layout

The contributions provided with the book samples are laid out using the default Maven project structure. For example, if you look at contributions/introducing-trips you'll see the following:

```
contributions/
    introducing-trips/
        src/
            main/
                java/
                    all the Java source code goes here
                resources/
                    all the non-Java resources go here
        build.xml – the Ant build script for the contribution
        pom.xml – the Maven build script for the contribution
```

Some contributions also have an src/test directory that hold artifacts to unit test the contribution.

To build the component service we first design the service interface. In this case the interface must allow a trip's availability to be checked and, if the trip is available, return a booking reference number.

The service interface is a normal Java interface, as shown in the following listing, and can be found in the sample's contribution/introducing-trips contribution directory.

Listing 1.1 The `Trips` interface definition

```
package com.goodvaluetrips;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface Trips {
    String checkAvailability(String trip, int people);
}
```

Note here that there's a curious `@Remotable` annotation. This is an SCA annotation that indicates that services implementing this interface will be accessible remotely over protocols such as SOAP/HTTP web services.

To make the TripProvider component do something, we have to provide some business logic in the form of a component implementation. This is an ordinary Java class that implements the `Trips` interface and returns a dummy booking code, as follows.

Listing 1.2 The implementation class for the TripProvider component

```
package com.goodvaluetrips.impl;

public class TripProviderImpl implements Trips {
    public String checkAvailability(String trip, int people) {
        return "6R98Y";
    }
}
```

All the Java language coding required to implement the TripProvider component is now done. The next step is to create a composite application and use the Tuscany runtime to run this component.

Tuscany doesn't run components directly but instead runs composite applications. An XML file called a composite file describes a composite application. We've already talked about components and the services they provide. A composite file describes the components that an application is built from and how they're wired together. In this case the composite file (named trips.composite) has only one component in it, as shown here.

Listing 1.3 The trips.composite file

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://goodvaluetrips.com/"
    name="Trips">

    <component name="TripProvider">
        <implementation.java
            class="com.goodvaluetrips.impl.TripProviderImpl" />
        <service name="Trips">
            <interface.java interface="com.goodvaluetrips.Trips" />
        </service>
    </component>
</composite>
```

At the root of this simple XML file is the `<composite>` element, which provides the composite name and some namespace declarations. Within the `<composite>` element the `<component>` element tells the Tuscany runtime that the application has a component called `TripProvider`. Detail of the component's Java implementation is provided by the `<implementation.java>` element. With this information the Tuscany runtime can create the component and provide the Trips service for others to call. SCA definitions typically refer to a service using the form componentName/serviceName,

because a single component can provide many services. In this case the service is referred to as TripProvider/Trips.

Composite applications are packaged into what SCA calls contributions. A contribution collects the composite file and the implementation and interfaces for the components that it describes. The GoodValueTrips contribution is made up of the compiled source code and associated resources from the resources directory. Contributions can be represented as directories on disk or as archives such as JAR or zip files. In this case we're going to use the target/classes directory on disk where the source is compiled to. It has the following layout:

```
trips.composite
com/
  goodvaluetrips/
    Trips.class
    impl/
      TripProviderImpl.class
```

This contribution contains the trips.composite file, the `Trips` interface, and the `TripProviderImpl` implementation class and can be found in the following sample directory: contributions/introducing-trips/target/classes.

All that's now required is the code that will start the runtime, load the GoodValueTrips contribution, and send a message to the TripProvider component. The following listing shows some simple Java code from the launchers/jumpstart directory that will do that for us.

Listing 1.4 Load the GoodValueTrips contribution and test the TripProvider component

```
package scatours;
import org.apache.tuscany.sca.node.SCAClient;
import org.apache.tuscany.sca.node.SCAContribution;
import org.apache.tuscany.sca.node.SCANode;
import org.apache.tuscany.sca.node.SCANodeFactory;

public class JumpstartLauncher {
  public static void main(String[] args) throws Exception {
   SCAContribution gvtContribution =
      new SCAContribution("introducing-trips",
        "../../contributions/introducing-trips/target/classes");
    SCANode node = SCANodeFactory.newInstance().
      createSCANode("trips.composite", gvtContribution);

    node.start();

    Trips tripProvider =
      ((SCAClient)node).getService(Trips.class, "TripProvider/Trips");

    System.out.println("Trip booking code = " +
      tripProvider.checkAvailability("FS1APR4", 1));

    node.stop();
}
}
```

1 Create contribution structure

2 Create node for contribution

3 Start the node

4 Get service proxy from node

5 Call the service

6 Stop the node

The contribution name and location are stored in the `SCAContribution` structure `gvtContribution` ❶. The contribution location can be either a relative path, if the contribution is located conveniently with respect to where the launcher is being run, or a full URL. This structure will be used to create the `SCANode` ❷, which is the Tuscany class that's used to control and access the SCA runtime. You'll note that when the node is constructed, the composite filename is also provided. This tells Tuscany to run the composite defined in the trips.composite file from `gvtContribution`. Once the node has been created, it needs to be started ❸ and is then ready for use.

At ❹ you'll use the `SCAClient` API on the node to retrieve a Java language proxy to the service named TripProvider/Trips, which is the Trips service associated with component TripProvider. You'll then use the returned proxy to call the service and check trip availability ❺. The resulting booking code will be printed out to the console. Finally you'll stop the node ❻ and the program will end.

When you run the `JumpstartLauncher` class, the application should print out:

```
Trip booking code = 6R98Y
```

The `JumpstartLauncher` class can be found in the following sample's launchers directory: travelsample/launchers/jumpstart/src/main/java/scatours.

If you were watching carefully, you may have noticed that the `Trips` interface used in listing 1.4 is defined in the `scatours` package and so isn't the same (in Java language terms) as the `Trips` interface in the `com.goodvaluetrips` package that we showed in listing 1.1. SCA provides more flexibility than the Java language by treating interfaces as equivalent if they contain the same operations, and `JumpstartLauncher` takes advantage of this SCA feature. The following listing shows the definition of the `Trips` interface used by `JumpstartLauncher`.

Listing 1.5 The jump-start launcher's `Trips` interface definition

```
package scatours;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface Trips {
    String checkAvailability(String trip, int people);
}
```

This example is one of the most basic things you can do with Tuscany, and on the face of it, it's not that interesting. It would have been easier to write the Java code and not bother with creating an SCA component. But now that you know how to build a component, we can explore what Tuscany is useful for, and that's assembling composite applications.

1.3.2 Defining more complex components

Our travel application has three components in all: TripProvider (which we've just looked at), TripBooking, and ShoppingCart. The artifacts for the TripBooking and ShoppingCart components can be found in the following sample contribution directory:

travelsample/contributions/introducing-tours. The TripBooking and ShoppingCart components are defined in the tours.composite file, shown here.

Listing 1.6 The tours.composite file

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://tuscanyscatours.com/"
    name="Tours">

    <component name="TripBooking">                          (1) TripBooking component
        <implementation.java
            class="com.tuscanyscatours.impl.TripBookingImpl" />
        <service name="Bookings">
            <interface.java interface="com.tuscanyscatours.Bookings" />
        </service>
        <reference name="mytrips" target="TripProvider/Trips">
            <interface.java interface="com.goodvaluetrips.Trips" />
        </reference>
        <reference name="cart" target="ShoppingCart/Updates">
            <interface.java interface="com.tuscanyscatours.Updates" />
        </reference>
    </component>

    <component name="ShoppingCart">                         (2) ShoppingCart component
        <implementation.java
            class="com.tuscanyscatours.impl.ShoppingCartImpl"/>
        <service name="Checkout">
            <interface.java interface="com.tuscanyscatours.Checkout" />
        </service>
        <service name="Updates">
            <interface.java interface="com.tuscanyscatours.Updates" />
        </service>
    </component>
</composite>
```

As with the earlier trips.composite file, the `<composite>` element in this tours.composite has an `xmlns` attribute indicating that its schema definition is in the XML namespace http://www.osoa.org/xmlns/sca/1.0. The contents of this namespace are defined in the SCA 1.0 specifications produced by Open SOA. The `<composite>` element also specifies a target namespace http://tuscanyscatours.com/ and a composite name of `Tours`. Together these define an XML QName for this composite. SCA requires all composite names to have a namespace qualification, which helps prevent accidental collisions with similar names elsewhere in the SCA domain. Well discuss the SCA domain in chapter 3, but for now imagine the SCA domain as a set of deployed composites that define the execution environment for one or more composite applications.

The `<composite>` element contains the definitions for the TripBooking (1) and ShoppingCart (2) components. Figure 1.7 shows a graphical representation of the TripBooking and ShoppingCart components.

Let's look at the details of the ShoppingCart component first. From the `<implementation.java>` element in the composite definition, you can see that this

component is implemented by the Java class `com.tuscanyscatours.impl.ShoppingCartImpl`. The component defines two services named Checkout and Updates, and each of these services specifies a Java interface that defines the signatures of the operations that it provides.

It's worth considering whether this component needs to have two different services. They could be combined by merging their interfaces, and at first glance this might seem like the simpler approach. The reason they're separate is because the Checkout service is used by external customers, and the Updates service is used internally for interactions between different parts of the travel-booking application. These services are in the same component because they share data within the component implementation.

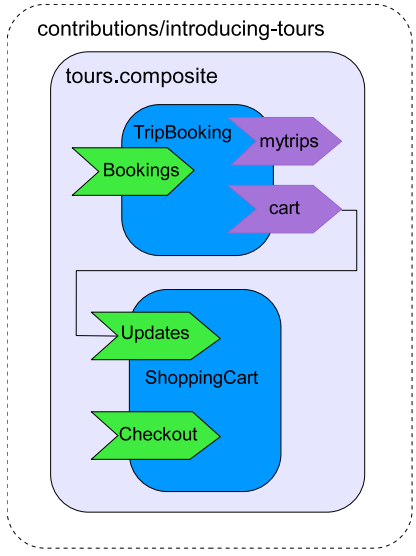


Figure 1.7 The TripBooking and ShoppingCart components shown wired together inside the Tours composite

Next we'll look at the TripBooking component definition. The component implementation class is `com.tuscanyscatours.impl.TripBookingImpl`. It defines one service named Bookings, and, for the first time, we see two references named `myTrips` and `cart`. As with services, references have interfaces that define the signatures of the operations that they can invoke. Both these references also have `target` attributes. These specify the service that the reference is wired to in the composite application. The service is identified by its component name and service name, separated by a / character. Here, the `cart` reference is wired to the Updates service provided by the ShoppingCart component. This means that operation calls made by the TripBooking implementation code using the `cart` reference in this code will be routed by SCA to the Updates service. Similarly, the `mytrips` reference is wired to the Trips service of the TripProvider component.

Component, service, and reference naming conventions

The SCA specifications don't dictate what conventions to use when naming components, services or references other than to define the rules about how service and reference names are derived if you don't specify them explicitly. In the travel sample, component and service names start with an uppercase letter and reference and property names start with a lowercase letter.

You already know how the TripProvider component is defined, but it's interesting to note that, unlike the TripBooking component, the TripProvider component has a service but no references. This pattern is often seen when an SCA component is used to "wrap" an existing application. The component implementation can use the existing application's APIs directly with no need to define SCA references.

It's important to point out that the implementation code for TripBooking doesn't know that its `mytrips` reference is connected to the TripProvider/Trips service. It also doesn't know where the TripProvider component is defined or how it's implemented. This is an example of the separation between implementation and configuration that we mentioned earlier.

You may be wondering why it's important to have this separation. In the travel business, companies come and go, and the best prices or most attractive packages may not always be available from the same provider. At the moment, TuscanySCATours is using the GoodValueTrips company as its trip provider. Next month, it might find another company, for example, BudgetTours, that offers cheaper rates for the same packages and provides a service TourProvider/Tours with a compatible interface. By changing the `mytrips` reference definition to the following,

```
<reference name="mytrips" target="TourProvider/Tours">
    <interface.java interface="com.goodvaluetrips.Trips" />
</reference>
```

TuscanySCATours could change its trip provider from GoodValueTrips to BudgetTours, without any need to modify or recompile any application code! Only the value of the target attribute has been changed from TripProvider/Trips to TourProvider/Tours; everything else remains the same.

In contrast, if our application code contained calls to a runtime for a specific communication technology such as web services or JMS, it's likely that changes to the application would be needed in order to move to a different trip provider. With features like this we're beginning to see the power and simplicity of SCA and Tuscany compared with other approaches such as explicit use of web services, and there's much more to come as we further explore SCA's capabilities.

1.3.3 Creating component implementations

We have completed our component definitions, and our next step will be to create implementations for the TripBooking and ShoppingCart components that we've defined. Doing things in this order is often referred to as top-down design and development, in contrast to the bottom-up approach we took when we wrote the TripProvider component, where we started with the implementation code first. SCA supports both approaches equally well. Typically the bottom-up approach is used when reusing existing code.

We'll begin by looking at the Java interfaces for the services Bookings, Checkout, and Updates. These are shown in listings 1.7, 1.8, and 1.9, respectively.

Listing 1.7 The `Bookings` interface definition

```
package com.tuscanyscatours;
import org.osoa.sca.annotations.Remotable;
@Remotable
public interface Bookings {
    String newBooking(String trip, int people);
}
```

Listing 1.8 The `Checkout` interface definition

```
package com.tuscanyscatours;
import org.osoa.sca.annotations.Remotable;
@Remotable
public interface Checkout {
    void makePayment(BigDecimal amount, String cardInfo);
}
```

Listing 1.9 The `Updates` interface definition

```
package com.tuscanyscatours;
import org.osoa.sca.annotations.Remotable;
@Remotable
public interface Updates {
    void addTrip(String resCode);
}
```

All these interfaces have the `@Remotable` annotation, which means that they describe SCA services that can be invoked from a different computer or a different process. Apart from the addition of the `@Remotable` annotation, these again are regular Java interfaces.

Now let's look at the implementation code for the TripBooking component.

Listing 1.10 The implementation class for the TripBooking component

```
package com.tuscanyscatours.impl;
import org.osoa.sca.annotations.Reference;
import com.goodvaluetrips.Trips;
import com.tuscanyscatours.Bookings;
import com.tuscanyscatours.Updates;

public class TripBookingImpl implements Bookings {
    @Reference
    protected Trips mytrips;

    @Reference
    protected Updates cart;

    public String newBooking(String trip, int people) {
        String resCode = mytrips.checkAvailability(trip, people);
        cart.addTrip(resCode);
```

SCA @Reference annotation

SCA @Reference annotation

```
        return "GV" + resCode;
    }
}
```

The important thing to notice here is the two SCA `@Reference` annotations. They identify the `mytrips` and `cart` variables as holding references to other SCA services. SCA requires these variable names to match the reference names of the component definition in the composite file. You'll see that there's no code to initialize the contents of these variables. This is because the Tuscany SCA runtime initializes them by injecting proxies for the services that were configured as wire targets for these references in the component definition.

Next we'll look at the implementation code for the ShoppingCart component.

Listing 1.11 The implementation class for the ShoppingCart component

```
package com.tuscanyscatours.impl;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;
import com.tuscanyscatours.Checkout;
import com.tuscanyscatours.Updates;

public class ShoppingCartImpl implements Checkout, Updates {
    private static List<String> bookedTrips = new ArrayList<String>();

    public void makePayment(BigDecimal amount, String cardInfo) {
       System.out.print("Charged $" + amount + " to card " +
                        cardInfo + " for trips" );
        for (String trip : bookedTrips){
             System.out.print(" " + trip);
       }
       System.out.println();
        bookedTrips.clear();
    }

    public void addTrip(String resCode) {
        bookedTrips.add(resCode);
    }
}
```

Because the ShoppingCart component provides two services, this implementation class needs to implement two interfaces, one for each service provided.

It's worth admitting now that this isn't a good shopping cart component. There's no obvious way that the shopping cart is associated with a particular customer. There's also a static field, bookedTrips, being used to store booked trips. But this is a sample, and the component improves later in the book.

It's also worth pointing out that we haven't needed to use much of the SCA programming model in this code! The only constructs that have been shown so far that weren't basic Java language constructs are the matching `@Remotable` and `@Reference` annotations in the service interface definitions and reference variables of component

implementations. What's missing from this picture? There isn't a single SCA-related API call in sight. Although programming models usually come with a sizable library of APIs, SCA has only a few APIs. In this simple SCA application we don't need to use any APIs. SCA's declarative programming style makes SCA easy to learn and use.

To complete the picture, we need to make some observations about the implementation of the TripProvider component that you saw in the previous section. The code to handle trip reservations was created by GoodValueTrips to service its clients (other travel businesses) and wasn't written using SCA. Because TuscanySCATours wants to access this code using SCA, the GoodValueTrips developers created the TripProvider component as a wrapper SCA component for this non-SCA code. In this pattern, an SCA-enabled implementation class delegates through to the non-SCA code by making direct Java language calls.

Using a wrapper component isn't the only way for SCA code to call non-SCA code. Another approach is to use remote bindings, for example, the web services or the RMI binding, to connect SCA components to applications running outside an SCA domain. We'll discuss these different approaches a little later in this chapter.

You've seen how SCA components can be implemented using regular Java language code with a small number of SCA annotations in a declarative programming style. With the Tuscany SCA runtime, these annotations cause reference values to be set by injection so that business logic doesn't need to use SCA API calls. This approach keeps business logic free from infrastructure concerns as far as possible and makes it easy for developers to create SCA component implementations with a minimum of SCA-specific code.

The next step in building our travel-booking application is to create a new SCA composite (Tours) to contain the two components that we've just defined and implemented. In the next section you'll see how to do this.

1.3.4 Wiring components to form a composite application

You've seen that SCA component configurations are always defined within composites. In its simplest form, a composite represents a grouping of one or more SCA components. Composites are defined in XML, using schemas that are part of the SCA specifications, and these XML definitions are placed in files with a .composite extension. In the initial travel-booking application there are two composite files: the trips.composite file, which you saw in listing 1.3, and the tours.composite file, which you saw in listing 1.6. The TripBooking and ShoppingCart components are defined by TuscanySCATours in the tours.composite file. The trips.composite file, which you saw in the jumpstart section, defines the TripProvider component separately from the TripBooking and ShoppingCart components because it's provided by GoodValueTrips. GoodValueTrips is a third-party travel provider that arranges and manages the trips sold not only by TuscanySCATours but also by other retail travel agents. Defining the TripProvider component in a separate composite makes it easier for other customers of GoodValueTrips to deploy and use it.

To deploy and run our initial travel-booking application, we'll need to create a package of the tours.composite file and its component implementations. You'll see what this involves in the next section.

1.3.5 Deploying a composite application using contributions

A *contribution* is a means of packaging together any set of items into a convenient unit that SCA and Tuscany can handle, and it can take a number of different forms. For example, it could be a zip or JAR file, or it could be a directory tree on the filesystem.

SCA doesn't impose any restrictions on what a contribution can contain. For example, it could contain WSDL files or JSP pages in addition to implementation code and component configurations. In chapter 3 you'll see some examples of doing this. An SCA application can be divided into multiple contributions with dependencies between them. This approach is useful when different parts of the application are developed independently and then deployed together to create a complete application. The initial travel-booking composite application is divided into two contributions to keep implementation code developed and owned by the TuscanySCATours company separate from code developed and owned by the GoodValueTrips company. We consider these two contributions to be part of the same composite application because the services that they define, via composite files, interact to satisfy the travel-booking business need. This relationship is notional though, because the two contributions could potentially be deployed and reused separately.

You saw the layout of the GoodValueTrips contribution in section 1.3.1. Here's the layout of the contents of the TuscanySCATours contribution:

```
tours.composite
com/
  tuscanyscatours/
    Bookings.class
    Checkout.class
    Updates.class
    impl/
        TripBookingImpl.class
        ShoppingCartImpl.class
  goodvaluetrips/
    Trips.class
```

This contribution can be found in the following sample directory: contributions/introducing-tours/target/classes. It contains the tours.composite file and six Java classes, four of which contain interface definitions (`Bookings`, `Checkout`, `Updates`, and `Trips`) and two others containing implementation code (`TripBookingImpl` and `ShoppingCartImpl`).

You might have noticed that the `Trips` interface appears in both the GoodValueTrips contribution and the TuscanySCATours contribution. But the `TripProvider` implementation class appears only in the GoodValueTrips contribution. This is because the `mytrips` reference in the TuscanySCATours contribution needs the interface but doesn't need the implementation.

It's interesting to pose the question of whether or not the TuscanySCATours contribution and the GoodValueTrips contribution are part of the same application. The answer could be yes or no, depending on what you mean by an application. The TuscanySCATours contribution can't function without the GoodValueTrips contribution, which suggests that they're part of the same application. But the GoodValueTrips contribution doesn't need the TuscanySCATours contribution, which suggests that they're separable. From this you see that the world of services is different from the world of applications, and it needs a different approach to packaging. Some services depend closely on other services, and it makes sense to package them together. Other services are more independent and are best packaged separately so that they can be reused in different contexts. SCA contributions are flexible enough to support both of these approaches with their ability to import dependencies from other contributions using SCA-specific import and export descriptions.

You can now run the initial travel-booking application. In this initial application sample you're going to skip the part where you design the user interface and connect it into the TripBooking and ShoppingCart components. This is covered in detail in chapter 8. For now you'll drive our application using a simple SCA client component that you wire to both the TripBooking and ShoppingCart components. You can find the client contribution in the following sample directory: contributions/introducing-client. The complete sample application configuration is shown in figure 1.8.

The application configuration in figure 1.8 should look familiar because it's mostly the same as the application configuration we first showed you in figure 1.5. The use of the TestClient component allows you to run the application without having to build a

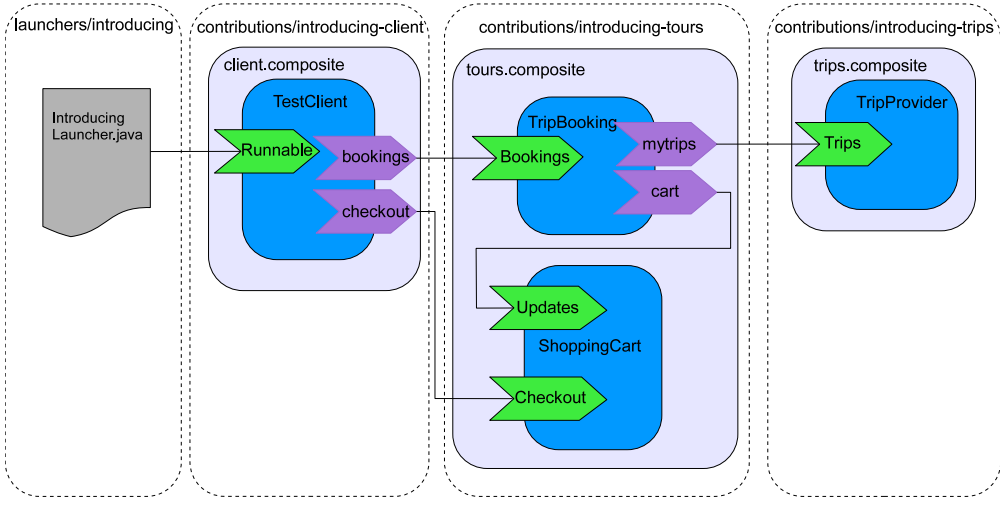


Figure 1.8 A test client component being used to test the TripBooking and ShoppingCart components we've built

user interface. The implementation of the TestClient component is simple and is shown here.

Listing 1.12 The TestClient component implementation

```
package scatours.client.impl;
import java.math.BigDecimal;
import org.osoa.sca.annotations.Reference;
import org.osoa.sca.annotations.Service;
import com.tuscanyscatours.Bookings;
import com.tuscanyscatours.Checkout;

@Service(Runnable.class)
public class TestClientImpl {
    @Reference
    protected Bookings bookings;

    @Reference
    protected Checkout checkout;

    public TestClientImpl() {
    }

    public void run() {
        String bookingCode = bookings.newBooking("FS1APR4", 1);
        System.out.println("Booking code is " + bookingCode);

        checkout.makePayment(new BigDecimal("1995.00"),
                             "1234567843218765 10/10");
    }
}
```

The `TestClientImpl` class simulates the calls to the SCA services that would be made as a result of the user selecting a trip from the browser and paying for it. It provides a single service named Runnable with the Java interface `java.lang.Runnable`.

Again you use a simple launcher Java program, which can be found in the following directory: launchers/introducing/src/main/java/scatours. The launcher loads the application's contributions into a node. Alternatively, the same code could be packaged as a JUnit test case, and we've provided a JUnit4 version of the launcher in the test directory of launchers/introducing. This time the `run` method of the TestClient component is retrieved and called. The following listing shows the code for the launcher program.

Listing 1.13 Load and test trips.composite and tours.composite

```
package scatours;
import static scatours.launcher.LauncherUtil.locate;
import org.apache.tuscany.sca.node.SCAClient;
import org.apache.tuscany.sca.node.SCANode;
import org.apache.tuscany.sca.node.SCANodeFactory;

public class IntroducingLauncher {

    public static void main(String[] args) throws Exception {
```

```
        SCANode node =
            SCANodeFactory.newInstance().createSCANode(null,
                                                locate("introducing-tours"),
                                                locate("introducing-trips"),
                                                locate("introducing-client"));

        node.start();

        Runnable proxy = ((SCAClient)node).getService(Runnable.class,
                                                      "TestClient/Runnable");
        proxy.run();

        node.stop();
    }
}
```

This launcher is slightly different from the `JumpstartLauncher` we showed back in listing 1.4. Here you use a `locate()` utility function to find each contribution. We created this function especially for the book sample so that the launchers can be run from the binaries directory as well as from the launcher source directory of the distribution without change. Take a look at the `LauncherUtil` class in the `util/launcher-common` module if you want to see what it does.

When you run the `IntroducingLauncher` class, you see the following output:

```
Trip booking code = GV6R98Y
Charged $1995.00 to card 1234567843218765 10/10 for trips 6R98Y
```

You've now learned how to build an SCA application packaged into contributions. Within the contributions are composites that define components, services, and references, together with implementation code for the components and interfaces for the services. That's all you need!

Now you know at a high level why SCA is a strong technology for developing SOA solutions. In the next section we'll help you understand how Tuscany and SCA can integrate with and complement other SOA-related technologies.

1.4 Working with other SOA technologies

You've seen how simple it is to build SCA components using Java classes and wire them together. You may now be asking the following questions:

- What if I don't want to use the Java language to implement my components?
- How do I integrate with my existing enterprise SOA technologies?

In this section we take a wider view to answer these questions. SCA specifications define how some of the popular technologies such as web services and BPEL fit with SCA. It would be impossible to cover integration with all SOA-related technology in this book. Instead, we describe three general approaches that SCA uses to allow you to exploit just about any technology you're faced with:

- API wrapping
- SCA implementations
- SCA remote bindings

Which approach you choose greatly depends on the features of the technology you're trying to integrate with.

1.4.1 API wrapping

We already touched on API wrapping in section 1.3.1. The TripProvider component, implemented by the GoodValueTrips company, was used to wrap an existing application by calling that application's APIs directly. Many applications and infrastructure technologies provide an API that can be used in this way.

When the technology you need to integrate with provides such an API, you can call that API from within a component implementation. As a reminder, here's the implementation class for the TripProvider component that you saw in the jump-start.

```
public class TripProviderImpl implements Trips {
    public String checkAvailability(String trip, int people) {
        return "6R98Y";
    }
}
```

Instead of directly returning the string `"6R98Y"`, we should be showing the code using the API that the GoodValueTrips company has designed to provide access to its existing application:

```
public class TripProviderImpl implements Trips {
    public String checkAvailability(String trip, int people) {
        GVTTrip gvTrip = GVTTripFactory.newTrip();
        gvTrip.setTripCode(trip);
        gvTrip.setNumberOfTravellers(people);
        GVTTripEngine.checkTrip(gvTrip);
        String bookingCode = gvTrip.getBookingCode();
        return bookingCode;
    }
}
```

In reality we've just invented this API here to demonstrate this point, but hopefully you appreciate that any Java language code could appear here.

1.4.2 Using SCA implementations

The Tuscany runtime provides support for integration with the Java language, BPEL, Java EE, and Spring, as defined by the SCA specifications. It also supports some Tuscany-specific integration such as widget, which supports integration with HTML and Web 2.0 client applications such as Ajax or Flex, and script for integrating with scripting languages such as Ruby, Groovy, and JavaScript. We'll cover these in chapters 5, 6, and 8.

We've already shown how the <implementation.java> element allows you to describe SCA component implementations using Java classes. You may also build an SCA component based on other implementation languages. This is as simple as using the appropriate implementation element for the component definition in the composite file.

For example, Spring is a popular technology for building enterprise applications. Tuscany can make good use of existing or newly created Spring application contexts.

Here's a contrived composite file, which doesn't actually exist in the travel sample, describing a TravelBooking component whose implementation is a Spring application context. In this case, one of the beans in the application uses a reference property called `payment`. This is wired, using the `target` attribute, to the Payment component that's implemented using a BPEL process. We'll cover how Spring and BPEL work with SCA in more detail in chapter 6.

```
<composite ...>

    <component name="TravelBooking">
        <implementation.spring location="./travel-booking-context.xml"/>
        <reference name="payment"
                   target="Payment/paymentPartnerLink" />
    </component>

    <component name="Payment">
        <implementation.bpel process="pp:Payment"/>
        <service name="paymentPartnerLink"/>
    </component>

</composite>
```

This shows how the Tuscany runtime uses the SCA implementation extensibility model to integrate with other technologies. It also shows that, most of the time, components are more complex than a single Java class file.

Now that you understand how to use different implementation technologies to create components, let's look at how you can enable components to communicate with each other or with external non-SCA services over remote bindings.

1.4.3 Using SCA remote bindings

The Tuscany Java SCA runtime supports binding extensions that allow your SCA components to interact with other services, either SCA or non-SCA, over a range of communications protocols. The SCA specifications define extensions for web services and JMS, and Tuscany supports these. Tuscany also provides support for other popular communication protocols such as RMI, JSON-RPC, and CORBA. These are all discussed in chapter 7.

Component implementations developed using SCA aren't aware of the protocol used to communicate with other components. The protocol of choice is attached to component references and services by adding bindings to the component definitions in the composite file. In our initial travel-booking application we didn't show how the ShoppingCart component interacted with the existing payment system. It could have been done in the following way:

```
<component name="ShoppingCart">
    <implementation.java
        class="com.tuscanyscatours.impl.ShoppingCartImpl"/>
    <service name="Checkout">
        <interface.java interface="com.tuscanyscatours.Checkout" />
```

```
    </service>
    <service name="Updates">
        <interface.java interface="com.tuscanyscatours.Updates" />
    </service>
    <reference name="payment">
        <binding.ws uri="http://slickpayments.com/paymentgateway"/>
    </reference>
</component>
```

Note that a `<binding.ws>` element has been added to the `payment` reference of the ShoppingCart component. The ShoppingCart component implementation doesn't know that communication with the payment application will take place using web services. The beauty of SCA is that this choice of binding can easily be changed to JMS by replacing the `<binding.ws>` element with the `<binding.jms>` element.

Some people have asked us about the difference between SCA and remote protocols like web services. This is a simple question to answer. SCA defines how a network of services can be assembled into usable applications and provides a language for describing this assembly. On the other hand, web services describe how to use a protocol to call an individual service.

SCA and Tuscany provide choice. If you want to expose or consume services as web services, you can do that. If you want to expose or consume services in other ways, for example, using JSON-RPC or RMI, you can do that too without changing the way you build your services. Choosing one binding doesn't preclude the use of other bindings either at the same time or in the future.

In the last few sections we covered how Tuscany works with various technologies. The next section addresses the question of how Tuscany aligns with Enterprise Service Bus (ESB) technology that's often present in an SOA implementation.

1.4.4 Tuscany and an Enterprise Service Bus

We've often been asked to compare Tuscany and SCA with the idea of the Enterprise Service Bus that's regularly used in SOA implementations. It comes up often enough that it's worth spending a little time on it here.

An ESB is typically used to allow software components to find and invoke each other irrespective of the technology used in their implementations. The ESB doesn't generally describe the composite application as a whole. It sits between components and manages message transfers from one component to another. In contrast, Tuscany and SCA allow you to describe your entire composite application.

If you already use an ESB, Tuscany is well placed to integrate with the ESB to exploit its interservice message routing and transformation capabilities. The system configuration is then described as an SCA assembly, and, where appropriate, the details of message transfer are managed by the ESB. For example, when you need to communicate via the ESB, you could configure a remote binding, such as the web services binding, to send messages to the ESB rather than directly to another component.

Clearly Tuscany doesn't require the use of an ESB. It's already specifically designed to allow you to assemble components into working applications. If you require

complex routing and message transformation in your composite application and you don't have or don't want to use an ESB, you can easily include components in your composite application that perform transformation and routing functions.

1.5 Summary

In this chapter we've talked about SCA and Tuscany, built a sample application, and looked at how to integrate SCA composite applications with other technologies. In particular we've explained how a typical SOA environment can be complex because of the range of technologies that need to work with one another. Apache Tuscany provides the means to control this complexity by offering a consistent end-to-end composite application model.

SCA achieves this consistency by providing a clear and concise description of how services are wired together, or composed, to form working applications. This so-called Assembly Model provides the tools to compose and recompose new and existing services into flexible applications as business requirements change.

SCA simplifies application development because its Assembly Model separates infrastructure concerns from business logic. This separation-of-concerns concept is central to how Tuscany is able to address SOA complexity and provide the developer with the following abilities:

- Choose the language used to implement each component
- Assemble components of different implementation types into composite applications
- Choose and alter the communication protocol bindings used to connect components together without modifying component implementations
- Choose the interface style used to describe a component's service and reference interfaces
- Choose the data format of a service or reference interface and have Tuscany automatically transform to and from the data format of other components
- Declaratively control nonfunctional application behavior using policy configuration or functional application behavior using properties

Tuscany achieves these capabilities by exploiting existing technologies. The Tuscany developer uses the SCA Assembly Model to describe how these existing technologies come together into a working application. The Tuscany runtime handles what's required to make the technologies work with one another.

By now you have a good high-level understanding of what SCA and Tuscany are all about. At this stage you needn't worry if you're not quite sure about the difference between a composite and a contribution! We'll come back to all these concepts in later chapters and cover them in much more detail, as well as introduce other advanced features of SCA and Tuscany that can make your business applications even more powerful and flexible. Now it's time to move on and start exploring in detail what Apache Tuscany has to offer. We start by looking more closely at how to define SCA components.

2 Using SCA components

This chapter covers

- Implementing SCA components
- Defining services, references, and properties
- Connecting components using wires and bindings

In the previous chapter we gave a practical introduction to Apache Tuscany and the major technical concepts in SCA, showing how a business application can use SCA and benefit from its capabilities. In this chapter and the next, we'll cover the essential concepts of SCA in more detail and show you how to use them to develop applications using Tuscany.

As a first step toward building Tuscany applications, you need to be familiar with SCA *components*, which are the fundamental building blocks for all SCA applications. We start by looking at the relationship between components and their implementations. Next, we'll show how components provide services themselves and invoke other services using references and wires. We'll examine how properties are used to customize aspects of a component's behavior. We will then show how bindings are used to select wire protocols for a component's services and references, and we'll finish the chapter by looking at how deployment of components into domains affects the choice of bindings.

You'll find plenty of examples and illustrations that show how to use these features of SCA in real applications, with discussions of design alternatives and guidance for best practice. The code samples in this chapter illustrate specific aspects of SCA using scenarios related to the simplified travel-booking application that we introduced in chapter 1. They're not used directly as part of the complete running travel application. You can find these samples in the contributions/usingsca directory in the travel sample code.

After reading this chapter, you'll understand how to use SCA components to create and customize applications that run on Apache Tuscany.

2.1 Implementing an SCA component

Every component contains an *implementation* that provides the component's business logic. For example, a component implementation could be a Java class, a BPEL process, or an EJB. The implementation controls what the component does and how it interacts with other components. Figure 2.1 shows a component implemented using a Java class and another component implemented using a BPEL process.

Each component has an *implementation type*, which identifies the implementation technology used by the component (for example, a Java class or a BPEL process). You can take advantage of SCA's support for different implementation types by choosing the implementation technology best suited to each application component. SCA components all have the same architecture, so it's easy to combine them into a composite application. For example, the travel-booking application that we described in chapter 1 uses components with Java implementations. In chapter 6 we'll show how you can add a BPEL component to this application. Components aren't aware of each other's implementation type, so a Java component can use a BPEL component without needing to know that it's written in BPEL, and vice versa.

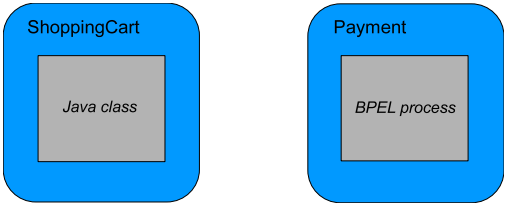


Figure 2.1 An SCA component implemented using a Java class and another SCA component implemented using a BPEL process

The SCA approach to components is different from other component technologies such as EJB, Microsoft .NET, and Spring. In SCA, any container-specific dependencies are encapsulated in the implementation type instead of being part of the component definitions. This is illustrated in figure 2.2.

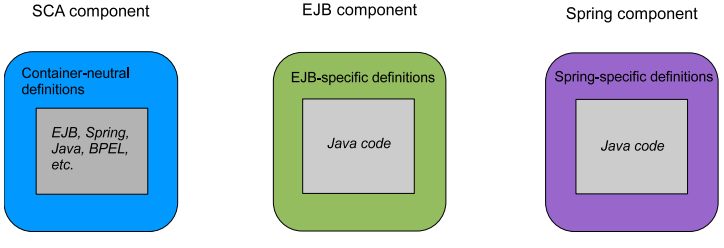


Figure 2.2 SCA component definitions aren't tied to any specific container technology, unlike components defined by other technologies such as EJB and Spring.

In the rest of this section we'll look at the different implementation types that you can use with SCA components, and we'll show how SCA uses component definitions to configure component implementations. We'll also take a detailed look at the SCA concept of component type, which defines the formal relationship between a component implementation and its component definition.

2.1.1 Choosing an implementation type

Each component definition specifies the implementation type of the component. For example, a component with a Java class as its implementation has an implementation type of `implementation.java`, and a component implemented by a BPEL process has an implementation type of `implementation.bpel`. SCA components are defined using XML, and the following example shows how the components pictured in figure 2.1 could be defined.

```
<composite name="cart"
           ....>
    <component name="ShoppingCart">
        <implementation.java
            class="com.tuscanyscatours.impl.ShoppingCartImpl" />
        ....
    </component>
    <component name="Payment"
               xmlns:pp="http://www.tuscanyscatours.com/Payment">
        <implementation.bpel process="pp:Payment" />
        ....
    </component>
</composite>
```

The `name` attribute of the `<component>` element specifies the component name. This is used to refer to the component and qualify references to its constituent parts. The `<composite>` element is used to enclose component definitions. A composite can't contain more than one component with the same name.

The ShoppingCart component has a Java implementation and is defined with the `<implementation.java>` element. The `class` attribute of this element contains the fully qualified name, `com.tuscanyscatours.impl.ShoppingCartImpl`, of the implementation class. The Payment component is implemented by a BPEL process, so it has an `<implementation.bpel>` element with a `process` attribute naming the BPEL process `Payment` in the XML namespace corresponding to the `pp` prefix. By convention, implementation elements are written as `<implementation.*>` with the `*` replaced by the specific implementation type.

The SCA 1.0 specifications define standard implementation types for Java code and code written in BPEL, C++, C, and COBOL, as well as Spring, web applications (.war files), EJBs, and Java EE archives (.ear files). SCA also provides an extension point for creating additional implementation types.

Tuscany provides implementations of the SCA implementation types for the Java language, BPEL, C++, Spring, and web applications. Tuscany also supports additional

implementation types that aren't currently part of the SCA specifications, enabling application developers to use the SCA component model with the following popular technologies:

- HTML pages containing Web 2.0 user interface widgets and JavaScript code, described in chapter 8.
- Scripting languages such as Python, JRuby, Groovy, and JavaScript. See chapter 6 for more details.

See chapters 5, 6, and 8 for more information about the implementation types in Tuscany. For application-specific needs that aren't covered by the Tuscany implementation types, application developers can create additional implementation types using extension points in the Tuscany runtime. Chapter 14 describes these extension points and shows how they can be used to create user-defined implementation types.

We've looked at the different implementation types that you can use for components. Next we'll show how to use SCA component definitions to configure implementations in different ways.

2.1.2 Configuring SCA components using component definitions

In SCA, a component is a configured implementation. The same implementation can be used in different components, configured in different ways. This enables code reuse while providing a high degree of flexibility and customizability. The configuration applied to an implementation that makes it into an SCA component is called a *component definition*.

SCA component definitions are written using an XML syntax. These definitions can configure those aspects of the component implementation that the implementation has exposed as being configurable. The configurable aspects of an implementation are collectively known as the implementation's *component type*. The component type includes the services, references, and SCA properties defined by the implementation. The implementation's business logic can't be modified using component definitions, so the business logic isn't part of the component type.

A hardware analogy for SCA components

To illustrate the relationship between the component implementation, the component type, and component definitions, we can use the analogy of a hardware component such as the 555 Timer IC (http://en.wikipedia.org/wiki/555_timer_IC). This chip is manufactured by different companies (Philips, Motorola, RCA, and so on) with different part numbers, and all versions of it work according to the same specification and can be used in the same electronic circuit. It's also possible for a circuit that uses this chip to customize its operation by varying the voltage or capacitance applied to various pins.

In SCA terms, the manufacturer's part number is an SCA component implementation, with different implementations possible for the same component functionality. When an individual chip is installed in a circuit, this is comparable to a configured SCA

(continued)

component, with the design of the surrounding circuit providing the component's configuration. The SCA component type corresponds to the chip's pin connections, which control the customizations that can be applied to the component implementation. The analogy isn't perfect, but it may help you to visualize what's going on when we talk about the component type of an SCA component implementation being configured by a component definition.

We'll use an example to show how the same SCA component implementation can be configured in different ways by different components. In this example, a car rental implementation defines an SCA property for the minimum age of the renter. This implementation can be configured by different car rental partner companies so that one company's car rental component requires a minimum age of 21 and the other company's component requires a minimum age of 25. This scenario is illustrated in figure 2.3, which shows how the same car rental implementation has been configured to produce a component for a provider of value cars with a lower minimum renter age and a component for a provider of luxury cars with a higher minimum age. Figure 2.3 shows the relationships between the implementation, the definitions of the two components, and the resulting components.

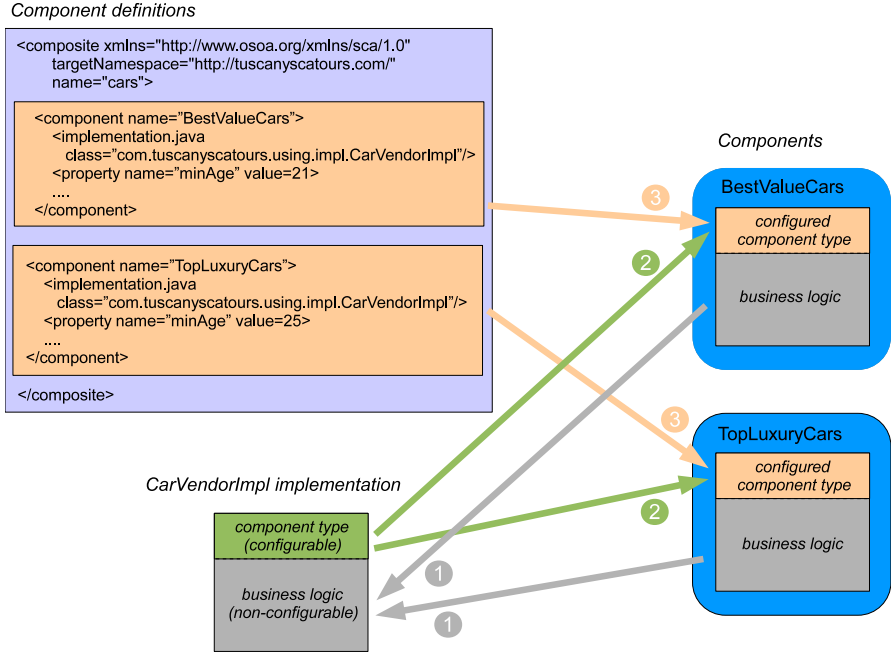


Figure 2.3 An SCA component implementation that's been configured using two different component definitions to produce two SCA components. The numbered arrows represent different kinds of relationships between the objects shown in the figure.

At the lower left of figure 2.3 is the CarVendorImpl implementation. For the purposes of the diagram, this is shown separated into its configurable portion (the component type) and its nonconfigurable portion (the business logic), represented by different sizes of rectangle. Above this are XML component definitions for BestValueCars and TopLuxuryCars with different configurations of the CarVendorImpl implementation. At the right of the figure are the SCA components BestValueCars and TopLuxuryCars that we've created from this implementation and these component definitions.

The component type isn't the same as the component's implementation type that we mentioned in section 2.1.1. The implementation type just says what kind of implementation technology the component uses, such as a Java class or a BPEL process. The component type is a much more detailed picture of the "shape" of this particular implementation, focusing on services, references, and properties, because these are the aspects of the implementation that can be configured by SCA component definitions.

The BestValueCars and TopLuxuryCars components share the same business logic from the CarVendorImpl implementation (shown by the arrows marked 1 in the figure). In contrast, each component has its own configuration resulting from merging its component definition with the implementation's component type (shown by the arrows marked 2 and 3 in the figure), with the component definition taking priority. As well as the difference in minimum renter age, the TopLuxuryCars component would have a different list of car types and prices than the BestValueCars component.

You've seen how the configurable shape of an implementation is represented by its component type and how SCA components are created by configuring component implementations using component definitions. Next, we'll look at what kinds of things are in the component type and how the component type of an implementation is defined.

2.1.3 Discovering or defining the component type

SCA uses two different approaches for discovering or defining the component type of an implementation, depending on whether the underlying implementation technology provides facilities for programmatically examining its own implementations (known as *introspection*) . For implementation types that support introspection (such as Java classes and BPEL processes), SCA defines rules for discovering the component type of an implementation by introspecting it. For implementation types where automatic discovery by introspection isn't possible, the definition of the component type must be provided alongside the implementation as a separate XML file.

For Java implementations, the component type is introspected from the component's Java implementation class. The following listing shows a simple example of a Java implementation class with no SCA annotations.

Listing 2.1 A Java component implementation class with no SCA annotations

```
package com.tuscanyscatours;

public class AirportCodes {
```

```
    public String getAirport(String code) {
        if ("AAA".equals(code)) return "Anaa";
        else if ("AAB".equals(code)) return "Arrabury";
        // other airport codes and cities would follow here
        else return null;
    }
}
```

You can find this example in the contributions/usingsca directory of the travel sample.

There's no need for the user to provide a component type definition because Tuscany determines the component type by introspecting the implementation class. The introspected component type for this class contains information that's equivalent to the following XML definition:

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
    <service name="AirportCodes">
        <interface.java interface="com.tuscanyscatours.AirportCodes" />
    </service>
</componentType>
```

Because the component type is determined by the Tuscany runtime using introspection, there's no physical artifact containing the above XML. In these cases, when we use the XML syntax in this book, it's as a human-readable way of showing the implementation's component type using a standard notation.

In this example there are no SCA annotations in the implementation class. If the class does contain SCA annotations, these are recognized and processed as part of introspecting the class to discover its component type, as you'll see in the following sections.

Some implementation technologies don't support introspection. SCA handles this situation by requiring a "side file" to be provided alongside the implementation to define its component type. This file has the extension .componentType and contains XML definitions written using the syntax shown previously. For SCA implementation types that use introspection such as `implementation.java`, side files aren't recognized and will be ignored if present.

We've looked at how component types are discovered and defined, and we've shown how a component type can be described using an XML representation. In the following sections we'll examine the details of what a component type can contain.

2.2 Using components to provide services

SCA is all about enabling service composition. Most SCA components provide at least one *service*, though it's possible to have components that use other services without providing any services themselves. These services can be used by other SCA components or by non-SCA code.

In this section we'll explore services in detail, starting with how services are defined by a component implementation and going on to explain how these services can be configured using component definitions. Finally, we'll look at some advanced features of service interfaces in SCA, such as local and remotable interfaces, bidirectional interfaces and callbacks, and conversational interfaces.

2.2.1 Defining services

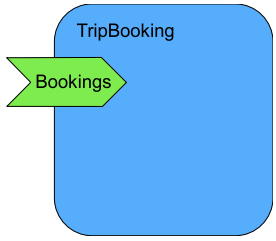


Figure 2.4 TripBooking is an SCA component with one service named Bookings.

The SCA specification describes a service as an addressable interface of the component's implementation. To put this into somewhat simpler language, every service has a name and an interface. The name identifies the service and distinguishes it from other services. The interface defines the names, inputs, and outputs of the operations that the service provides. Figure 2.4 shows a component with one service.

The services provided by a component are defined by the component's implementation. The details of how these services are defined depend on the component's implementation type. For example, when using `implementation.java`, the `@Service` annotation on the service implementation class specifies one or more Java interfaces that represent services provided by the implementation. The following example shows how `@Service` can be used:

```
package com.tuscanyscatours.usingsca.impl;
import com.tuscanyscatours.Bookings;
import org.osoa.sca.annotations.Service;

@Service(Bookings.class)
public class TripBookingImpl implements Bookings {
    ....
    public String newBooking(String trip, int people) {
        ....
    }
}
```

This implementation defines a single service named Bookings with the Java interface `com.tuscanyscatours.Bookings`. This interface could be defined as follows:

```
package com.tuscanyscatours;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface Bookings {
    String newBooking(String trip, int people);
}
```

All the methods of the interface are exposed by the service.

If the implementation class isn't annotated with `@Service`, SCA assumes that the implementation class has services corresponding to all its implemented interfaces that are annotated with `@Remotable` (see section 2.2.4 for more information about remotable services). The `Bookings` interface is annotated with `@Remotable`, so we could have omitted the `@Service` annotation from the implementation class.

The component implementation class is usually declared as implementing (in the Java language sense) its service interfaces. In the previous example, the `TripBookingImpl` class does this by declaring that it implements the `Bookings` interface.

However, SCA doesn't insist on this and only checks that all the methods of the service interfaces are present in the implementation class. You can find full details of how to define services for Java implementations in chapter 5.

Other implementation types have different ways of defining services. For example, `implementation.bpel` introspects the BPEL process to find partner links that receive a message before sending a message and maps these partner links to SCA services.

One ring to bind them all...

If you're used to working with other component models, you may find it surprising that SCA delegates the task of service definition to component implementations and allows each implementation type to do this in its own way. This approach works because SCA requires all implementation types to support the same service and component semantics. These semantics are embodied in the SCA definition of component type. This consistency ensures that all SCA services and components can be configured and composed in the same way, no matter what implementation technology was used to define them. The same goes for references and properties, as you'll see in later sections of this chapter. The combination of a common abstraction for components and services together with complete flexibility for how this abstraction is expressed in different implementation technologies is one of the most powerful and compelling features of SCA.

Every service has an *interface* that specifies the operations provided by the service and the input and output types of those operations. In the previous example, the SCA service Bookings has the Java interface `com.tuscanyscatours.Bookings`. This means that the Bookings service has a single operation, `newBooking`, with inputs of a `String` and an `int`, and a `String` as output. A service with a Java interface containing two methods would have two operations. Let's look at how interfaces are defined in SCA.

2.2.2 Interface definition in SCA

Interfaces are widely used in many programming and communication technologies. For example, a Java interface defines a programming contract that can be provided by an implementation class and is used by other implementation classes. C++ uses abstract base classes for the same purpose. In Web Services, WSDL 1.1 definitions use portTypes (renamed to interfaces in WSDL 2.0) to define the operations that can be used to interact with services. CORBA uses OMG IDL (Interface Definition Language) to define the interface provided by a remote object. These are just a few examples of existing interface-definition technologies. SCA has chosen not to create an interface-definition mechanism of its own but instead to make interface definition an extensibility point so that existing styles of interface definition can be used for SCA services.

Why doesn't SCA have its own interface definition language?

An important part of SCA's vision is to provide a service-oriented programming model that can be used with the broadest possible range of existing technologies. SCA can be used to compose Java implementations, web service endpoints, CORBA distributed objects, and much more. When connecting Java implementations, it's natural to use Java interfaces. For composing web services, WSDL definitions are the preferred choice, and when exposing a CORBA implementation using SCA, it's easiest to use CORBA IDL interfaces that have already been defined. A new interface definition language imposed by SCA would not add value but would make it harder for users to create SCA services that use these technologies.

The interface languages that can be used to define SCA interfaces are represented in SCA by different *interface types*. For example, a service whose interface is defined by a WSDL 1.1 portType is represented in an SCA component definition by the `interface.wsdl` interface type and XML element. Here's an example:

```
<interface.wsdl interface=
    "http://tuscanyscatours.com/#wsdl.interface(Bookings)" />
```

This defines an SCA interface using a WSDL 1.1 portType named `Bookings`, which is in the XML namespace http://tuscanyscatours.com/. In case you're wondering about using WSDL 2.0, this is permitted in SCA 1.0 but isn't supported by Tuscany and has been removed from SCA 1.1.

An SCA service with a Java interface would have the `interface.java` interface type, as in the following example:

```
<interface.java interface="com.tuscanyscatours.Bookings" />
```

Here the SCA interface is defined using the Java interface whose fully qualified name is `com.tuscanyscatours.Bookings`. The `<interface.wsdl>` and `<interface.java>` elements both contain an `interface` attribute, with each interface type specifying what this attribute value can contain.

What happens when SCA is used to compose services that use different interface technologies? For example, the Trips service could be implemented as a web service endpoint defined in WSDL, and the Bookings service, which uses Trips, could have a Java implementation. We'd like to be able to define the Trips service with a WSDL 1.1 portType for its interface, while allowing the Bookings service to use a Java interface when referring to the same service. SCA supports these combinations, which brings up some interesting questions, such as what to do about overloaded methods (allowed in Java interfaces but not in WSDL) and how to handle data transfer objects (Java interfaces use Java objects, but WSDL uses XML schema types). You'll see how SCA deals with this when we look at wiring in section 2.3.2.

You've seen how services and interfaces are defined in SCA. Now we'll look at how an SCA component definition can customize the definitions of the component's services.

2.2.3 Configuring services in component definitions

The component definition can specify configuration settings for services defined in the implementation's component type. For example, the component definition can change a service interface to use a different interface language. The Bookings service that you saw in section 2.2.1 has an introspected interface type of `interface.java`. The interface type could be changed to `interface.wsdl` in a component definition, which means that the component service interface is a WSDL 1.1 portType. This could be useful when creating a component that will be accessed by other components that don't have Java implementations. Here's an example component definition that does this:

```
<component name="WebBooking">
    <implementation.java
          class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
    <service name="Bookings">
        <interface.wsdl interface=
            "http://tuscanyscatours.com/#wsdl.interface(Bookings)" />
    </service>
</component>
```

Another example of component service configuration is a component definition overriding the component implementation's service interfaces to subset the service operations defined by the component implementation. The component definition can't add additional operations or additional services that go beyond those defined by the component implementation.

A component definition can't remove any of the services defined by its component implementation. We'll show how you can subset these services in section 3.4.2.

If the component definition doesn't specify any configuration for the service, the default configuration computed by the component type discovery rules is used. For example, the same implementation could be used in another component defined as follows:

```
<component name="TripBooking">
    <implementation.java
          class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
</component>
```

This component definition doesn't mention the service or its interface, so it has a Bookings service with an interface definition taken from the implementation's introspected component type. We're using a Java implementation type, and this means that the introspected interface type will always be `interface.java`.

You've seen how implementations define services and how services use interfaces to specify the operations that the service provides. Services are part of the implementation's component type, which represents the configurable aspects of the implementation. SCA supports a number of different implementation types as well as different interface types. Component definitions are used to configure components by combining component implementations with additional configuration information. The component type represents those aspects of a component implementation that can be

What you see may not be what you get

When you look at a component definition, you may not see everything that the component contains. This is because all the services, references, and properties from the implementation's component type automatically become part of the component as is, unless they're reconfigured in the component definition. Some people feel slightly uncomfortable with this and prefer to spell out component definitions in full by repeating everything from the component type in the component definition.

reconfigured by a component definition. If you're comfortable with all of this, you're well on the way to understanding SCA! Now it's time to look at some other interesting things you can do with SCA interfaces.

2.2.4 Local and remotable interfaces

SCA supports *local* and *remotable* interfaces. A service with a local interface can only be called from code running in the same operating system process (a local call). A service with a remotable interface has the additional ability to be called from code running in a different process or on a different computer (a remote call). Services defined with remotable interfaces can be used for local calls as well as remote calls.

It's up to the definition of each SCA interface type as to whether interfaces of that type can be local, remotable, or both. Interfaces defined using the interface type `interface.wsdl` are always remotable. Interfaces defined using the interface type `interface.java` can be either local or remotable; the default is local, with remotable interfaces identified using the `@Remotable` Java annotation.

The choice of a local or remotable interface has consequences for implementation developers. With a local interface, parameters and results are passed by reference (shared), and any changes made to these shared programming language objects by the service implementation are visible to calling code. With a remotable interface, parameters and results are passed by value (copied), and any changes made by the service implementation aren't seen by the calling code. Sharing data can be more efficient, but it creates a closer dependency between implementation details of the calling code and the service implementation (tight coupling). Copying data means that calling code and service implementation are less dependent on each other's internal details (loose coupling), but the copying can be costly, especially if the amount of data being passed is large. The best of both worlds would be to have the flexibility of remotable interfaces without the cost of data copying, and SCA and Tuscany provide optimizations to support this for many common cases.

Some services need to be remotable because they're called by other services that are running in a different location. Remotable services also provide better deployment flexibility than local services, because a remotable service can be moved to a different execution host machine or process without affecting any code that calls the service. Moving a service might be necessary to improve scalability, fault tolerance, or

security. In contrast, a local service can't be moved unless all the code that calls the service is moved at the same time, so that the locality relationship between calling code and service implementation is maintained.

You can find more detailed information about local and remotable interfaces in chapters 4 and 5, with examples of how you can use them in Tuscany applications. Next we'll look at how SCA supports callbacks using bidirectional interfaces.

2.2.5 Bidirectional interfaces and callbacks

In most service-definition technologies, the roles of service provider and service requester are clearly distinguished. The service provides business functionality, and the service requester makes use of this functionality by invoking service operations defined by the service interface. In some cases a service interaction also requires the service provider to make invocations back to the service requester. To support this interaction style, SCA provides the ability to define a *bidirectional* interface.

A bidirectional interface is a combination of two interfaces: one that's used for service requesters to invoke the service provider and one that's used by the service provider to make invocations to the service requester (referred to as *callbacks*). In an interface definition, the forward interface for invocations from requester to service provider is specified using the `interface` attribute, and the callback interface for invocations from service provider to requester is specified using the `callbackInterface` attribute. Here's an example of a bidirectional interface definition for the `interface.java` interface type:

```
<interface.java
    interface="com.tuscanyscatours.common.Search"
    callbackInterface="com.tuscanyscatours.common.SearchCallback" />
```

A service requester using a service that provides a bidirectional interface needs to support the callback interface so that the service provider can make callbacks to the service requester. The details of how the requester's implementation provides this support depend on the requester's implementation type. For `implementation.java`, this support is normally provided by the requester's implementation class implementing the callback interface.

Bidirectional interfaces can be local or remotable. Mixing local and remotable interfaces within a bidirectional interface definition isn't allowed, so you can't specify a remotable callback interface for a local forward interface (or vice versa).

There's a detailed discussion of bidirectional interfaces and callbacks in chapter 4, with examples of business scenarios that require this interaction style and sample code showing how to implement the service requester and provider sides of a bidirectional interaction. Now we'll look at how service implementations can maintain state between service invocations by using conversational interfaces.

2.2.6 Conversational interfaces

Some interactions between service requesters and providers require a sequence of calls to perform a business function. For example, an initial call could create an order,

subsequent calls could add items to the order, and a final call could submit the order for processing. This kind of interaction is often described as stateful, because of the need for both the requester and provider to maintain internal state data reflecting the current state of the interaction.

SCA 1.0 supports the concept of *conversational* interfaces to model this kind of interaction style. For `interface.java`, the interface can be marked as conversational by using the `@Conversational` annotation on the Java interface. For `interface.wsdl`, SCA has defined the global attribute `requires` in the SCA namespace, which can be used to mark a WSDL 1.1 `<portType>` or WSDL 2.0 `<interface>` element as representing a conversational SCA interface.

Conversational interfaces can be local or remotable. It's possible to mark a bidirectional interface as conversational in its forward direction, in its callback direction, or in both directions. Chapter 4 contains a detailed explanation of conversational interfaces and shows how to use them in Tuscany applications.

> **NOTE** The future of conversational interfaces in the SCA specifications is under discussion. Although there are similar concepts in some other middleware technologies, there's currently no accepted industry standard for conversational semantics. Because of the lack of industry standardization in this area, conversations aren't included in the SCA 1.1 specifications being standardized by OASIS and will be reconsidered for a future version of SCA.

You've seen how services are defined and configured in SCA and the important role that interfaces play in defining services. Now we'll look at how SCA service implementations can make use of other services.

2.3 Connecting components using references and wires

Service implementations often need to use other services, and they identify their dependencies on other services by defining *references*. Let's suppose that TuscanySCA-Tours is changing its business model to partner directly with car, flight, and hotel providers instead of using a trip-provider service to purchase prepackaged trips. This reduces the cost of purchasing trips and also opens up the possibility of selling customized trips. To implement this we'll modify the TripBooking component so that it has references for a car-booking service, a flight-booking service, and a hotel-booking service, as shown in figure 2.5.

In this section we'll show how a component's references are defined and how they're connected to other services. We'll also show how a single reference can be connected to more than one service.

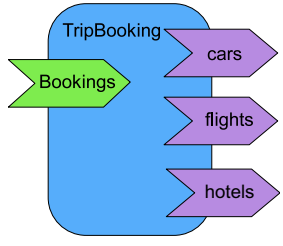


Figure 2.5 TripBooking is an SCA component with one service and three references.

2.3.1 Defining references

The details of how to define a reference vary between implementation types. For Java implementations, the `@Reference` annotation is used for this. The following example shows a reference definition for a flight-booking service:

```
package com.tuscanyscatours.usingsca.impl;
import com.tuscanyscatours.Bookings;
import com.tuscanyscatours.Flights;

public class TripBookingImpl implements Bookings {

    @Reference
    protected Flights flights;

    ....
}
```

In this example code, `Bookings` and `Flights` are Java interfaces for the trip-booking and flight-booking services, `TripBookingImpl` is the implementation class for the trip-booking service, and the `flights` field in the `TripBookingImpl` class is annotated with `@Reference`. The name of the reference defaults to `flights`, which is the name of the field. This default name can be overridden by adding a `name` member to the `@Reference` annotation. When an SCA runtime such as Tuscany creates an instance of the `TripBookingImpl` class, it injects a proxy for the flight-booking service into the `flights` variable. This means that the `flights` variable is initialized to the correct value, even though there's no code in `TripBookingImpl` that does this.

The injected proxy implements the same Java interfaces as the service that it calls. When called by code in the service requester, it invokes the service on the requester's behalf (hence the term *proxy*). For example, to call the flight-booking service to make a reservation, the requester could call the reference proxy injected into the `flights` field using code like the following:

```
flights.bookFlight(flightNumber, date, numSeats, bookingClass);
```

References can be injected into Java implementations using constructor parameters, fields, or setter methods. See chapter 5 for full details of how to do this. The following listing is an example showing these different styles of reference injection for Java implementations. You can find this example in the contributions/usingsca directory of the travel sample.

Listing 2.2 Reference injection using fields, setter methods, and constructor parameters

```
package com.tuscanyscatours.usingsca.impl;
import org.osoa.sca.annotations.Reference;
import com.tuscanyscatours.Bookings;
import com.tuscanyscatours.Cars;
import com.tuscanyscatours.Flights;
import com.tuscanyscatours.Hotels;
```

```
public class TripBookingImpl implements Bookings {

    public TripBookingImpl(@Reference(name="cars") Cars cars) {
        this.cars = cars;
    }

    protected Cars cars;

    @Reference
    protected Flights flights;

    private Hotels hotels;

    @Reference
    public void setHotels(Hotels hotels) {
        this.hotels = hotels;
    }

    ....
}
```

Implementation business logic goes here

In listing 2.2, a reference proxy to a car-booking service is injected using a constructor parameter, a reference proxy to a flight-booking service is injected using a field, and a reference proxy to a hotel-booking service is injected using a setter method.

The introspected component type contains three references named `cars`, `flights`, and `hotels`. It doesn't make any distinction among these references based on the different ways they were defined in the Java code (as a constructor parameter, a field, and a setter method). This is because the component type is an abstraction of the configurable shape of the implementation and doesn't include any Java-specific implementation details.

Other implementation types have their own ways of defining references. For example, the BPEL implementation type introspects the BPEL process to find partner links that send a message before receiving a message, and it maps these partner links to SCA references.

You've seen how component implementations can define references, and you've looked at different ways of injecting reference proxies into Java implementations. Now it's time to examine how references know which services to call.

2.3.2 Wiring references to services

The connection between a reference and a service is called a *wire*, and the process of making these connections is called *wiring*. Tuscany wires a component's references when the component is deployed, before the component's implementation starts executing.

Figure 2.6 shows how the TripBooking component could be wired to services for making car bookings, flight bookings, and hotel bookings. In this figure, the wires are the lines connecting references and services. Wiring is part of component configuration, and wires are specified using component definitions.

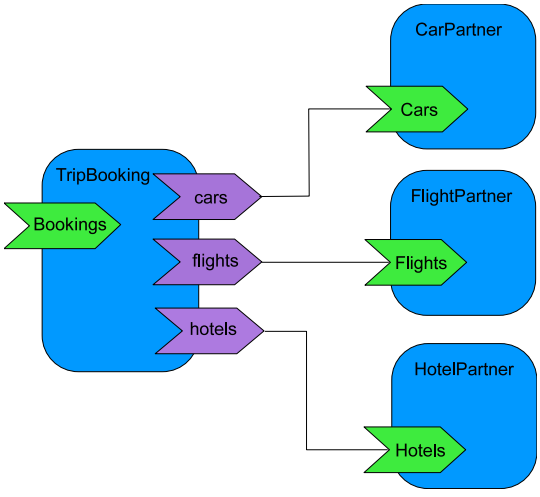


Figure 2.6 The SCA references in the TripBooking component have wires connecting them to SCA services in other SCA components: CarPartner, FlightPartner, and HotelPartner.

The following listing shows component definitions corresponding to the components and wires in figure 2.6. This example can be found in the contributions/usingsca directory of the travel sample.

Listing 2.3 Connecting component references and services using wires

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="bookings1">
    <component name="TripBooking">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
        <reference name="cars" target="CarPartner/Cars" />
        <reference name="flights" target="FlightPartner/Flights" />
        <reference name="hotels" target="HotelPartner" />
    </component>
    <component name="CarPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
    <component name="FlightPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.FlightPartnerImpl" />
    </component>
    <component name="HotelPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.HotelPartnerImpl" />
    </component>
</composite>
```

This XML file doesn't include any definitions for the component services. This is because the service definitions are taken from the component types of the component implementations. We're not doing any configuration of these services, so there's no

need to mention them in the XML file. We do need to mention the component references explicitly because we're providing wiring information for them.

References are wired to services using the `target` attribute of the `<reference>` element to specify the relative URI of the service. The syntax of this URI is a component name followed by a service name, separated by a `/` character. If the target component has a single service, it's sufficient to specify just the component name. We're using this shorthand when wiring the `hotels` reference to the Hotels service of the HotelPartner component.

When a reference is wired to a service, the interfaces of the reference and service must be compatible. This means that the service's interface needs to provide all the operations in the reference's interface, with the same input types, output types, and exceptions. The service could also provide additional operations that aren't used by the reference. The SCA specifications describe this as a "compatible superset" relationship. For a bidirectional interface, the operations on the reference's callback interface must be a compatible superset of the operations on the service's callback interface, because the roles of service requester and provider are reversed when making callbacks.

If the interfaces of the reference and service are defined using the same interface type, it's easy for Tuscany to look at the operation signatures to check compatibility. If the interfaces are defined using different interface types, one of the interface types needs to be mapped to the other interface type. (We mentioned this situation in section 2.2.2.) For example, if the reference's interface uses `interface.java` and the service's interface uses `interface.wsdl`, the SCA rules require Tuscany to use the Java-to-WSDL mapping defined by JAX-WS to convert the reference's interface to a WSDL 1.1 portType and compare this with the service's interface. In this case, the Java interface needs to be defined as remotable and isn't allowed to use method overloading. Java data types passed as parameters or return types on the methods of the reference's interface are converted to XML schema types using either the JAXB or SDO mapping. Chapter 9 describes data type mapping in more detail. If this interface mapping produces incompatible interfaces, or if the reference and service are defined using different interface types that don't have a mapping between them, the reference and service can't be wired together.

Using the `target` attribute on a reference isn't the only way to specify a wire. We'll look now at other ways of wiring references to services.

2.3.3 Wire elements

Instead of using a `target` attribute on the `<reference>` element, you can use a separate `<wire>` element to connect a reference to a service. The `<wire>` element has `source` and `target` attributes for the wire's reference and service. For example, the wire from the `hotels` reference of the TripBooking component to the Hotels service

of the HotelPartner component could be written as a <wire> element by making the following changes to the code in listing 2.3:

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="bookings2">
    <component name="TripBooking">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
        <reference name="cars" target="CarPartner/Cars" />
        <reference name="flights" target="FlightPartner/Flights" />
    </component>
    <wire source="TripBooking/hotels" target="HotelPartner" />
    ....
</composite>
```

In this version the <reference> element for hotels has been replaced by a <wire> element. A sample that uses this composite file can be found in the contributions/usingsca directory of the travel sample. The value of the <wire> element's source attribute is a component name and reference name separated by /. The reference name can be omitted if the component only has one reference. The <wire> element's target attribute specifies a target service using the same relative URI notation as the target attribute of the <reference> element. Putting all this together, the <wire> element in our example connects the TripBooking component's hotels reference (note the lowercase h) to the HotelPartner component's Hotels service (the only service for this component).

Using the <wire> element to specify a wire has exactly the same meaning as using the target attribute of the <reference> element. Because <wire> elements aren't part of component definitions, the wiring configuration is completely separate from the component definitions, allowing wiring changes to be made without affecting any component definitions. Another advantage of using this approach is that components can be deployed without being wired, and the wires connecting the components can be deployed separately. This provides deployment-time flexibility in how components are wired together.

So far we've assumed that every reference is explicitly wired to a single service. SCA also allows references to be wired automatically, wired to more than one service, or left unwired. We'll describe these possibilities in the remainder of this section. These SCA capabilities fall into a slightly more advanced category, so it's fine to skip ahead to section 2.4 if you just want to understand the basics of SCA.

2.3.4 Automatic wiring

As an alternative to explicit wiring using target attributes or <wire> elements, SCA has an *autowire* capability. This automatically wires references to services in the same composite that have a matching interface, without the need to specify <reference> targets or <wire> elements.

The following listing shows how the example in listing 2.3 could be rewritten using autowire. You can find this example in the travel sample's contributions/usingsca directory.

Listing 2.4 Using autowire to connect references and services

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="bookings3">
    <component name="TripBooking" autowire="true">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
    </component>
    <component name="CarPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
    <component name="FlightPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.FlightPartnerImpl" />
    </component>
    <component name="HotelPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.HotelPartnerImpl" />
    </component>
</composite>
```

In this version of the example, we've enabled automatic wiring for the TripBooking component by specifying `autowire="true"` on its `<component>` element. This means that Tuscany automatically wires every reference within the component to a service in the same composite that has a matching interface, as defined by the interface compatibility rules described in section 2.3.2. If there's more than one matching service in the composite, the choice among them is implementation dependent. In this example, each of the three references in TripBooking has exactly one matching service, so there's no ambiguity. If no matching services are found in the composite, an error is raised.

Automatic wiring can be a great time-saver, but it can cause problems if the composite happens to contain more than one service that matches a reference. In these cases, the automatic wiring rules might make a choice that the developer was not expecting, without any warning that this has happened!

Next we'll look at how SCA allows a reference to be wired to more than one service or left unwired.

2.3.5 Reference multiplicity

The `multiplicity` attribute of the `<reference>` element specifies the number of wires that can be used for the reference, which is referred to as the reference's *multiplicity*. The different settings are listed in table 2.1.

Table 2.1 Multiplicity settings for references with their meanings

Multiplicity setting	Meaning of the setting
`multiplicity="1..1"`	The reference is always wired to a single service.
`multiplicity="1..n"`	The reference is always wired to one or more services.
`multiplicity="0..1"`	The reference is optionally wired to a single service.
`multiplicity="0..n"`	The reference is optionally wired to one or more services.

The `n` in these multiplicity settings is a literal that appears exactly as shown. For example, it isn't possible to specify a multiplicity of `"1..5"`.

Wiring a reference to multiple services allows an implementation to make a choice at runtime between different providers of a service. For example, the CarPartner component from the example in section 2.3.2 could have the service and references shown in figure 2.7.

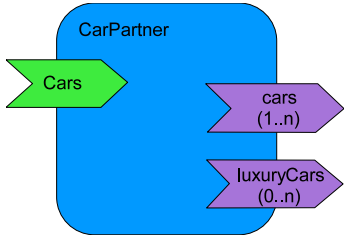


Figure 2.7 The CarPartner component has a reference `cars` with multiplicity `1..n` and another reference `luxuryCars` with multiplicity `0..n`.

In this figure, the `cars` reference has multiplicity `1..n`, which allows it to be wired to more than one car-vendor service. This enables the business logic in the CarPartner component to select a car vendor based on price and availability at the time each car booking is made. The CarPartner component needs at least one car vendor, so the `cars` reference is defined with multiplicity `1..n` rather than `0..n`. Some rental locations also have vendors specializing in luxury cars, and the `luxuryCars` reference can be wired to these vendors if there are any. Some rental locations have only regular cars and no luxury cars, so the `luxuryCars` reference is defined with multiplicity `0..n` to allow it to be left unwired. Here's the component type corresponding to figure 2.7:

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
    <service name="Cars">
        <interface.java interface="com.tuscanyscatours.Cars" />
    </service>
    <reference name="cars" multiplicity="1..n">
        <interface.java interface="com.tuscanyscatours.Cars" />
    </reference>
    <reference name="luxuryCars" multiplicity="0..n">
        <interface.java interface="com.tuscanyscatours.Cars" />
    </reference>
</componentType>
```

The multiplicity of references defined within implementations is determined by the rules for the implementation type. For Java implementations, the `@Reference` annotation can have a `required` member with a value of `false` to indicate that injection of the reference is optional. An array or collection used as a reference means that the reference is multivalued, with all available targets injected as elements of the array or collection. The default value of the `required` member is `true`.

See chapter 5 for more information and code examples showing how to define different multiplicities for references in Java component implementations. For components that don't have Java implementations, each implementation type defines its own way of specifying these multiplicity settings.

We've shown why references with different multiplicities are useful and how they can be defined. Next we'll look at how wiring works for different multiplicities.

2.3.6 Wiring with different multiplicities

Let's look at how wires are specified for references whose multiplicity isn't `1..1`. We'll illustrate these using the example wiring configuration in figure 2.8.

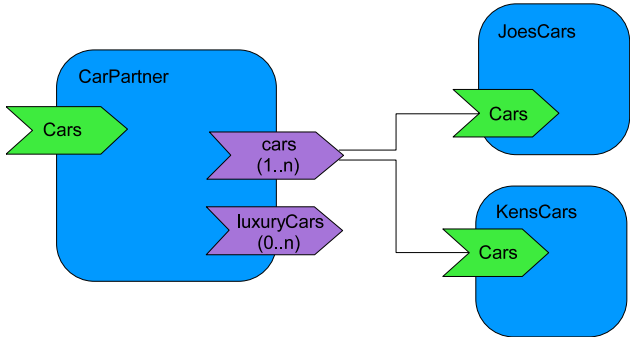


Figure 2.8 The `cars` reference of CarPartner is wired to two car vendor components: JoesCars and KensCars. The other reference, `luxuryCars`, is left unwired.

In figure 2.8 the CarPartner component is configured with a choice of two rental car vendors, Joe's Cars and Ken's Cars. This is shown by multiple wires from the `cars` reference of CarPartner. No vendors of luxury rental cars are available, so the `luxuryCars` reference has been left unwired.

We'll start by showing how this configuration can be defined using the `target` attribute of the `<reference>` element. The following code shows how you could write component definitions for the wiring setup shown in figure 2.8. This example can be found in the travel sample's contributions/usingsca directory.

Listing 2.5 Wiring references with different multiplicities

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="carbookings1">
    <component name="CarPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarPartnerImpl" />
        <reference name="cars" target="JoesCars KensCars" />
```

```
    </component>
    <component name="JoesCars">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
    <component name="KensCars">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
</composite>
```

Multiple targets for a reference are specified using a single `<reference>` element with space-separated names in the `targets` attribute. In listing 2.5 the `target` attribute of the `<reference>` element for `cars` specifies two target services. It's sufficient to just give the component names JoesCars and KensCars, because each of these components has only one service. There's no `<reference>` element for the `luxuryCars` reference, so this reference will be left unwired.

References with multiplicity can also be wired using `<wire>` elements. Each `<wire>` element can specify only one target, so we'll need a separate `<wire>` element for each target service. The following listing shows how you could rewrite the example from figure 2.8 using `<wire>` elements.

Listing 2.6 Wiring references with different multiplicities using `<wire>` elements

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="carbookings2">
    <component name="CarPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarPartnerImpl" />
    </component>
    <wire source="CarPartner/cars" target="JoesCars" />
    <wire source="CarPartner/cars" target="KensCars" />
    <component name="JoesCars">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
    <component name="KensCars">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
</composite>
```

In listing 2.6 the `<reference>` element from listing 2.5 has been omitted, and `<wire>` elements are used instead. You can find this code in the travel sample's contributions/usingsca directory.

The rules for autowire are slightly different when references have multiplicities other than `1..1`. If an autowired reference has multiplicity `0..n` or `1..n` (an upper bound greater than `1`), it's wired to all matching services in the composite. For references with a multiplicity of `0..1` or `0..n` (a lower bound of `0`), it isn't an error if no matching services are found. The next listing shows the example from figure 2.8 with

autowired references. This example is included in the travel sample's contributions/usingsca directory.

Listing 2.7 Wiring references with different multiplicities using autowire

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="carbookings3">
    <component name="CarPartner" autowire="true">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarPartnerImpl" />
        <reference name="luxuryCars" autowire="false" />
    </component>
    <component name="JoesCars">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
    <component name="KensCars">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
</composite>
```

In listing 2.7 we've taken advantage of autowire to eliminate the need to write targets or wires. The `cars` reference of CarPartner will be automatically wired to both of the Cars services in the JoesCars and KensCars components because these services have interfaces matching the `cars` reference. You'll notice that we've turned off autowire for the `luxuryCars` reference. This reference has the same interface as the `cars` reference, so the autowire rules would incorrectly wire this reference to the Cars services in JoesCars and KensCars. This is an example of the potential problems with autowire that we mentioned in section 2.3.4.

You've seen how references can be defined and how they can be connected to services in a number of different ways. Next we'll look at how components can be configured using properties.

2.4 Configuring components using properties

Implementations can enable external configuration of some of the values that they use in their processing. These externally configurable values are called *properties*. In this section you'll see how to define properties and how to configure values for both simple and complex property types.

Figure 2.9 shows a currency converter component with two properties.

We'll use this component as an example to show how properties are defined and how to configure values for properties.

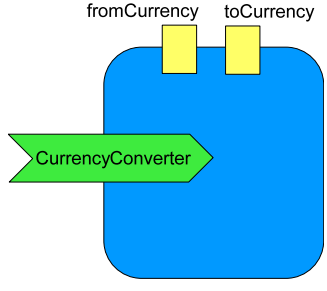


Figure 2.9 A currency converter component with two properties, `fromCurrency` and `toCurrency`

2.4.1 Defining properties

As with services and references, the way properties are defined depends on the implementation type. For Java implementations, the `@Property` annotation is used to define an SCA property. Listing 2.8 illustrates how this annotation can be used to define the `fromCurrency` and `toCurrency` properties for the currency converter component shown in figure 2.9. You can find this code in the travel sample's contributions/usingsca directory.

Listing 2.8 Defining properties for a component implementation

```
package com.tuscanyscatours;
import java.math.BigDecimal;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface CurrencyConverter {
    BigDecimal convert(BigDecimal amount);
}

package com.tuscanyscatours.usingsca.impl;
import java.math.BigDecimal;
import org.osoa.sca.annotations.Property;
import com.tuscanyscatours.CurrencyConverter;

public class CurrencyConverterImpl implements CurrencyConverter {{

    @Property
    protected String fromCurrency;

    @Property
    protected String toCurrency;

    public BigDecimal convert(BigDecimal amount) {
        return amount.multiply(getRate(toCurrency))
                     .divide(getRate(fromCurrency), 2, 0);
    }

    private BigDecimal getRate(String currency) {
        ....
    }
}
```

Code to get exchange rate goes here

Listing 2.8 shows a service interface `CurrencyConverter` and a component implementation class `CurrencyConverterImpl`. In the implementation class the `@Property` annotation has been applied to the `fromCurrency` and `toCurrency` fields. When the Tuscany runtime creates an instance of `CurrencyConverterImpl`, Tuscany injects property values into these fields. Like references, properties can be injected using constructor parameters, fields, or setter methods.

You've seen an example of how properties can be defined. Next you'll see how property values can be configured using component definitions.

2.4.2 Configuring values for properties

Property values are configured using `<property>` elements in the component definition. The `name` attribute of the `<property>` element identifies the property being configured. For simple property types such as strings and numbers, the `<property>` element's content is used as the value for the property in the configured component.

In our example each configured component converts between a particular pair of currencies, which are specified as the values of the `fromCurrency` and `toCurrency` properties. The following listing shows component definitions that create converters from EUR to JPY and from USD to GBP. This code is available in the travel sample's contributions/usingsca directory.

Listing 2.9 Configuring property values in component definitions

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="converter">
    <component name="EURJPYConverter">
        <implementation.java class=
            "com.tuscanyscatours.usingsca.impl.CurrencyConverterImpl" />
        <property name="fromCurrency">EUR</property>
        <property name="toCurrency">JPY</property>
    </component>
    <component name="USDGBPConverter">
        <implementation.java class=
            "com.tuscanyscatours.usingsca.impl.CurrencyConverterImpl" />
        <property name="fromCurrency">USD</property>
        <property name="toCurrency">GBP</property>
    </component>
</composite>
```

The `convert` method of EURJPYConverter converts euros into yen, and the `convert` method of USDGBPConverter converts dollars into pounds. For conversions between other currency pairs, it's easy to create additional converter components by specifying different values for the properties.

Like services and references, properties are part of the implementation's component type. The component type for CurrencyConverterImpl looks like this:

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <service name=CurrencyConverter">
        <interface.java
            interface="com.tuscanyscatours.CurrencyConverter" />
    </service>
    <property name="fromCurrency" type="xs:string" />
    <property name="toCurrency" type="xs:string" />
</componentType>
```

You might be looking at this example and wondering why we don't just define the currency codes as Java static constants. Creating a new converter would be a simple matter

of changing the values of the constants and recompiling the class. There are three main advantages of using SCA properties instead of Java constants:

- There's only one implementation of the class. Using Java constants would require making an extra copy of the implementation class with different constant values for each different currency conversion pair.
- Property configuration can be done without changing the code within the class. For most businesses, changing implementation code is expensive—even for a small change like this. This is why SCA provides so many facilities to adapt and change applications using external configuration rather than by modifying implementation code. The important thing isn't the amount of code change that's needed but whether changes can be made without needing to modify the implementation.
- A management tool can display and change the values of SCA component properties much more easily than values within implementation code. For small applications with a few components, this may not be important. When these components become part of a larger and more complex application, it's easy to see the manageability and configurability benefits of defining some carefully chosen properties.

Configurability, configurability, configurability

SCA is all about the ability to reconfigure various aspects of an application without going in and changing the code. Every feature of SCA is designed to make this easier for business developers and infrastructure developers. Tuscany provides a runtime framework for SCA that supports the same goals of flexible configuration and infrastructure extensibility.

We've shown how to define and configure properties that are simple scalars with string values. In the rest of this section we'll show how properties can use the full power of XML schema to represent complex data structures. This is a more advanced capability of SCA properties, and it's fine to skip ahead to section 2.5 if you just want to cover the basics now.

2.4.3 Using complex types for properties

Any XML schema complex type defined as a global element or schema type (that is, an element or schema type declared as a child of the schema root element) can be used as the type of an SCA property. Following is an example of an XML schema global element representing a U.S.-style address.

Listing 2.10 XML schema global element definition for SCA property

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://tuscanyscatours.com/">
```

```
    <xs:element name="billingAddress">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="street" type="xs:string" />
                <xs:element name="city" type="xs:string" />
                <xs:element name="state" type="xs:string" />
                <xs:element name="zip" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

You can use this as the type of a property in a component implementation. For example, a Customer component could be implemented using Java code similar to the following:

```
package com.tuscanyscatours;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface CustomerInfo {
    ....
}

package com.tuscanyscatours.usingsca.impl;
import org.osoa.sca.annotations.Property;
import com.tuscanyscatours.BillingAddress;
import com.tuscanyscatours.CustomerInfo;

public class CustomerImpl implements CustomerInfo {
    @Property
    protected BillingAddress billingAddress;
    ....
}
```

The `BillingAddress` class that's used as the Java type of the `billingAddress` property needs to be mappable to the XML schema global element shown in listing 2.10. For example, the `BillingAddress` class could be generated from the XML schema by using the JAXB mapping from XML schema to Java code. Listing 2.11 shows an example definition of the Customer component with a configured value for the `billingAddress` property. You can find all the code needed to run this example in the contributions/usingsca directory of the travel sample.

Listing 2.11 SCA configuration for property defined as global element

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace=" http://tuscanyscatours.com/"
    xmlns:t="http://tuscanyscatours.com/"
    name="orders1">

    <component name="Customer">
        <implementation.java
              class="com.tuscanyscatours.usingsca.impl.CustomerImpl" />
```

```
        <property name="billingAddress" element="t:billingAddress">
            <t:billingAddress xmlns="">
                <street>123 Main Street</street>
                <city>New York</city>
                <state>NY</state>
                <zip>01234</zip>
            </t:billingAddress>
        </property>
    </component>
</composite>
```

The `element` attribute of the `<property>` element indicates that the property type is defined as an XML schema global element. The property value is an instance of this global element, appearing as a child of the `<property>` element.

It's also possible to specify the property type using an XML schema type instead of a global element. This can be useful when multiple properties have the same type definition. For example, let's suppose that the `CustomerImpl` component implementation is extended to contain a delivery address as well as a billing address, where both of these addresses are defined by the XML schema type shown in the following listing.

Listing 2.12 XML schema type definition for SCA property

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace=" http://tuscanyscatours.com/">
    <xs:complexType name="Address">
        <xs:sequence>
            <xs:element name="street" type="xs:string" />
            <xs:element name="city" type="xs:string" />
            <xs:element name="state" type="xs:string" />
            <xs:element name="zip" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

The new property `deliveryAddress` is added to the `CustomerImpl` class as follows:

```
package com.tuscanyscatours.usingsca.impl;
import org.osoa.sca.annotations.Property;
import com.tuscanyscatours.Address;
import com.tuscanyscatours.BillingAddress;
import com.tuscanyscatours.CustomerInfo;

public class CustomerImpl implements CustomerInfo {
    @Property
    protected BillingAddress billingAddress;
    @Property
    protected Address deliveryAddress;
    ....
}
```

The new property `deliveryAddress` has the Java type `Address`, which is mapped from the schema type `Address` using the JAXB mapping. The Java type of the `billingAddress` property is `BillingAddress` (as used in the previous version of the code), which works because this Java type is mappable to the property schema type `Address`.

Because the billing and delivery addresses are defined in XML using a schema type, their values are configured in the component definition using `<property>` elements with `type` attributes instead of the `element` attribute used in listing 2.11. The property values are specified using an element of the `Address` schema type. For example, you could define an `<address>` schema global element as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://tuscanyscatours.com/orders"
           xmlns:t="http://tuscanyscatours.com/">
    <<xs:element name="address" type="t:Address" />
</xs:schema>
```

You can use this element to configure the property values for `billingAddress` and `deliveryAddress`. The full component definition is shown in the following listing, and the code for this example is available in the contributions/usingsca directory of the travel sample.

Listing 2.13 SCA configuration for properties defined using a schema type

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace=" http://tuscanyscatours.com/"
    xmlns:t="http://tuscanyscatours.com/"
    xmlns:o="http://tuscanyscatours.com/orders"
    name="orders2">

    <component name="Customer">
        <implementation.java
              class="com.tuscanyscatours.usingsca.impl.CustomerImpl" />
        <property name="billingAddress" type="t:Address">
            <o:address xmlns="">
                <street>123 Main Street</street>
                <city>New York</city>
                <state>NY</state>
                <zip>01234</zip>
            </o:address>
        </property>
        <property name="deliveryAddress" type="t:Address">
            <o:address xmlns="">
                <street>456 Market Street</street>
                <city>San Francisco</city>
                <state>CA</state>
                <zip>98765</zip>
            </o:address>
        </property>
    </component>
</composite>
```

We've shown how property values are specified using XML within component definitions. It's also possible to set a property value using the value of some other property or from the contents of a file. These are described in the SCA Assembly Model specification, which is available at http://osoa.org/display/Main/Service+Component+Architecture+Specifications.

We've looked at all the configurability points of an SCA component: services, references, and properties. Now it's time to look at how SCA services and references can be configured to control the wire-level protocols and endpoints that they use when they communicate.

2.5 Enabling communication flexibility using bindings

One of the most important features of SCA is its support for a wide variety of communication protocols. If your services need to talk Web Services, JMS, CORBA, RMI, or REST, they can do it using SCA and Tuscany. If they need to use some specialized or proprietary protocol to meet a particular application need, that's fine too. Even better, your business code doesn't need to know which protocol it's using; the choice of a protocol is made by (you guessed it) the component's configuration. How cool is that? The piece of SCA magic that makes all this possible is called a *binding*.

In the rest of this section you'll see how to use bindings on services and references and what it means if no bindings are configured. Finally, we'll look at the SCA domain and see how bindings relate to communication within and outside an SCA domain.

2.5.1 Configuring bindings for services and references

You can use bindings on services or references. Putting a binding on a service means that the service can be invoked using the communication protocol specified by the binding. The service implementation code doesn't need to do anything special to make this happen; all you have to do is add the binding to the service's configuration. Each binding has a *binding type* that identifies the communication technology used by the binding, such as Web Services or JMS.

For example, to make the Bookings service available as a web service, the component definition might look like this:

```
<component name="TripBooking">
    <implementation.java
          class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
    <service name="Bookings">
        <binding.ws uri="http://tuscanyscatours.com:8085/Bookings" />
    </service>
</component>
```

By adding the `<binding.ws>` element, you've told the SCA runtime that you want to use the SCA Web Services Binding to expose the Bookings service as a web service using SOAP/HTTP at the endpoint specified in the `uri` attribute. That's all you need to do to create a web service. There's no need to learn JAX-WS or to make any changes to the Bookings service's implementation.

A service can have more than one binding. If you want to invoke the Bookings service over JMS as well as Web Services, you just need to add another binding. Here's an example component definition showing this:

```
<component name="TripBooking">
    <implementation.java
          class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
```

```
    <service name="Bookings">
        <binding.ws uri="http://tuscanyscatours.com:8085/Bookings" />
        <binding.jms uri="jms:Bookings" />
    </service>
</component>
```

The `<binding.jms>` element makes the Bookings service available over JMS. Again, it wasn't necessary to learn the JMS API or change any implementation code.

Bindings provide the linkage between SCA and Tuscany and the rest of the world! The Bookings service is written in SCA and runs in Tuscany. Because this service is configured using bindings, it can be called by client code that isn't written using SCA or running in Tuscany. For calls to the Bookings web service, the client code can be written in any language and can use any WS-I–compliant Web Services runtime. For JMS calls, the client code can be written directly to the JMS API using any JMS provider that's compatible with the JMS provider configured for Tuscany's JMS binding.

You've seen how bindings can be used on SCA services to make them available using standard communication protocols. In the same way, bindings can be used on SCA references to call non-SCA services using a standard communication protocol. In this case the roles are reversed; the client of the non-SCA service is implemented using SCA, and the non-SCA service uses some standard communication protocol and can be implemented using any technology that supports the chosen protocol. For example, the service could be a web service endpoint that supports SOAP/HTTP, or it could be an EJB session bean communicating over RMI-IIOP. To talk to a web service, the reference would specify `<binding.ws>`, and for calling an EJB, you'd use `<binding.ejb>`.

The `binding.ws`, `binding.jms`, and `binding.ejb` binding types are defined by the SCA specifications and are provided as part of Tuscany. If your services need to use a protocol that Tuscany doesn't support out of the box, you can use Tuscany's support for extension points to create a new binding type for that protocol. Chapter 14 shows an example of how to do this.

Figure 2.10 shows a diagrammatic representation of configuring bindings on a service and references of a component named TripBooking. Listing 2.14 shows an example component definition for the service, references, and bindings pictured in figure 2.10. The code for this example is available in the contributions/usingsca directory of the travel sample.

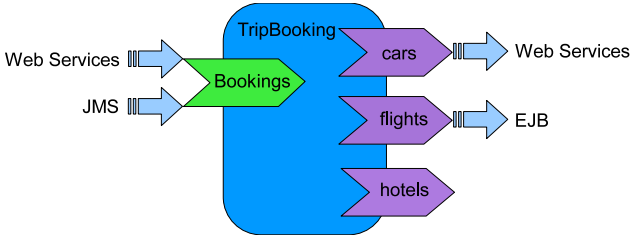


Figure 2.10 The Bookings service is configured with Web Services and JMS bindings, the `cars` reference is configured with a Web Services binding, and the `flights` reference is configured with an EJB binding.

Listing 2.14 Configuring component services and references with bindings

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="bookings4">
    <component name="TripBooking">
        <implementation.java
              class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
        <service name="Bookings">
            <binding.ws uri="http://tuscanyscatours.com:8085/Bookings" />
            <binding.jms uri="jms:Bookings" />
        </service>
        <reference name="cars">
            <binding.ws uri="http://tuscanycars.com:8081/Cars" />
        </reference>
        <reference name="flights">
            <binding.ejb uri="corbaname:rir:#flight/FlightPartnerHome" />
        </reference>
        <reference name="hotels" target="HotelPartner" />
    </component>
    <component name="HotelPartner">
        <implementation.java
            class="com.tuscanyscatours.usingsca.impl.HotelPartnerImpl" />
    </component>
</composite>
```

You'll notice that `<reference>` elements in listing 2.14 that are configured with bindings don't have a `target` attribute. This is because the binding provides the destination information for the reference.

We've looked at how bindings can be applied to services and references. Now we'll look at what happens for services and references that don't specify a binding in their configuration.

2.5.2 The default binding

In listing 2.14 there isn't a binding for the `hotels` reference. This means it has an implicit binding type of `binding.sca` (often called the *default binding*). The default binding is used to connect an SCA reference to an SCA service, leaving the choice of communication technology to the SCA runtime in which the reference and service are deployed. Other bindings (such as `binding.ws` and `binding.jms`) select a specific communication protocol or API that can interoperate with non-SCA code. Because of this, bindings other than the default binding are often called *interoperable bindings*.

Tuscany uses Web Services over SOAP/HTTP for remote calls over `binding.sca`; other SCA runtimes might use a different standard protocol such as RMI-IIOP, or they could use a proprietary protocol. In the future Tuscany could switch to something else for the default binding, so Tuscany applications shouldn't assume that using the default binding means they're getting Web Services. An application that wants to use Web Services to communicate between components should specify `<binding.ws>` to make sure this happens.

The default binding can be used only for connecting a reference to a service within the same SCA *domain*. Connections that span domain boundaries must specify an interoperable binding. The domain is an important concept in SCA, and we'll describe it in detail in chapter 3. For now, we'll give a brief overview of what an SCA domain is and describe how it relates to bindings and wiring.

2.5.3 Domains, bindings, and wiring

The SCA domain is a deployment and management boundary for SCA components. For example, a domain could be a single application server, a server cluster, or a group of servers or clusters. An entire domain typically runs a single vendor's implementation of SCA.

Every SCA component is part of an SCA domain. References and services within the same domain can be connected using wires. Connections between references and services within the same domain can use the default binding because SCA guarantees that the implementation of this binding will be consistent within a domain. For communication outside the domain, references and services can't use wires and instead must use interoperable bindings to interact with other services.

We'll illustrate the use of domains with a scenario based on the TuscanySCATours company that we looked at in chapter 1. The company has grown, and it has created a separate division to provide a specialized hotel-booking service. This hotel-booking service is used by the TuscanySCATours trip-booking service, and it's also available directly to customers who only want to book a hotel. The two divisions of the company use separate SCA domains to deploy and administer the services they provide: the TuscanySCATours domain for the original trip-booking service and the TuscanySCAHotels domain for the new hotel-booking service. Figure 2.11 illustrates this scenario.

In this figure, the solid lines connecting SCA components represent SCA wires, and the broken lines with arrows represent connections that use an interoperable binding instead of a wire. We'll start by looking at the connections from the references of the TripBooking component in the TuscanySCATours domain. The `flights` reference is connected to a web service that makes flight bookings. This isn't an SCA service, so the `flights` reference is configured with the interoperable Web Services binding and the endpoint URI of the flight-booking service. The `hotels` reference is connected to the Hotels service of the HotelPartner SCA component in the TuscanySCAHotels domain. Because SCA wires can't span domains, the reference and service are connected by configuring both of them with the Web Services binding and the service's endpoint URI. Finally, the `cars` reference of TripBooking is connected to the Cars service of the CarPartner component in the TuscanySCATours domain. Because the reference and service are in the same domain, we can connect them using an SCA wire and the default binding. The Cars service isn't exposed outside the TuscanySCATours domain and doesn't interoperate with non-SCA code, so there's no need for it to have any interoperable bindings.

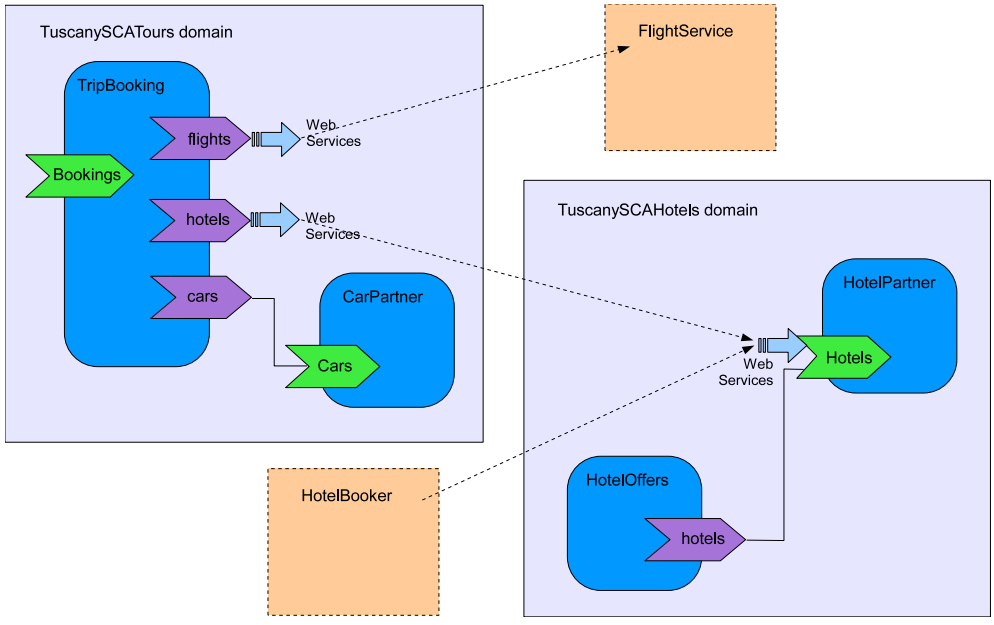


Figure 2.11 Two SCA domains containing components, showing how the components are connected. Wires (shown as solid lines) are used for connections inside each domain, and bindings (shown as broken lines with arrows) are used for connections that cross a domain boundary.

Now let's look at the connections to the Hotels service of the HotelPartner component in the TuscanySCAHotels domain. As well as the connection from the `hotels` reference of the TripBooking component, there are connections from HotelBooker (a hotel-booking client application that doesn't use SCA) and from the `hotels` reference of the SCA component HotelOffers in the TuscanySCAHotels domain. Because the Hotels service is configured using the interoperable Web Services binding, the HotelBooker software can use any web service client software to invoke the Hotels service. The connection between HotelOffers and HotelPartner uses an SCA wire because these components are in the same SCA domain. This wire could use either the default binding or the interoperable Web Services binding that the service provides. In this example we've chosen to use the default binding because this is the simplest approach for wires within an SCA domain. Another option would have been to use the Web Services binding at both ends of the wire.

It's useful to see how the connections in figure 2.11 could be configured using component definitions. In chapter 3 we'll cover domain configuration and composite deployment in detail; for now, the following listing shows an example of how the components and connections in the TuscanySCATours domain in figure 2.11 could be defined for a deployment configuration.

Listing 2.15 Component definitions for the TuscanySCATours domain

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="toursdomain">
    <component name="TripBooking">
        <implementation.java
              class="com.tuscanyscatours.usingsca.impl.TripBookingImpl" />
        <reference name="flights">
            <binding.ws
                uri="http://flightbookingservice.com:8084/Flights" />
        </reference>
        <reference name="hotels">
            <binding.ws uri="http://tuscanyscahotels.com:8083/Hotels" />
        </reference>
        <reference name="cars" target="CarPartner/Cars" />
    </component>
    <component name="CarPartner">
        <implementation.java
              class="com.tuscanyscatours.usingsca.impl.CarVendorImpl" />
    </component>
</composite>
```

The next listing shows example component definitions for the TuscanySCAHotels domain. You can find the code for listings 2.15 and 2.16 in the contributions/usingsca directory of the travel sample.

Listing 2.16 Component definitions for the TuscanySCAHotels domain

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscahotels.com/"
           name="hotelsdomain">
    <component name="HotelPartner">
        <implementation.java class=
          "com.tuscanyscatours.usingsca.impl.HotelPartnerImpl" />
        <service name="Hotels">
            <binding.ws uri=
                "http://tuscanyscahotels.com:8083/Hotels" />
            <binding.sca />
        </service>
    </component>
    <component name="HotelOffers">
        <implementation.java class=
          "com.tuscanyscahotels.impl.HotelOffersImpl" />
        <reference name="hotels"
                target="HotelPartner/Hotels" />
    </component>
</composite>
```

1 binding.ws for use outside the domain

2 binding.sca for use within the domain

3 this reference uses binding.sca

It's important to specify `<binding.sca>` 2 as well as `<binding.ws>` 1 on the Hotels service. Without this the `hotels` reference 3 couldn't be wired to the Hotels service using the default binding (`binding.sca`). Another option would be to put `<binding.ws>` on the `hotels` reference and omit `<binding.sca>` from the Hotels service.

What's in a name?

In the example code in this chapter you've seen various names in a number of different styles such as all lowercase (toursdomain) and concatenated capitalized words with either an uppercase initial letter (TripBooking) or a lowercase initial letter (minAge). Although SCA doesn't impose or recommend any particular naming convention, the following suggestions may be useful. The names of artifacts defined by a component (services, references, and properties) will usually follow the naming style of the component's implementation type. For SCA artifacts (components, composites, and domains), there's no recommended or established SCA naming convention. For XML-related artifacts (types, elements, and namespaces), the usual XML conventions apply.

The example in figure 2.11 shows most of the ways that SCA references and services can be connected. It's also possible to use a matched pair of bindings to connect an SCA reference to an SCA service in the same SCA domain, as an alternative to wiring. This could be useful if an SCA reference needs to invoke an SCA service that only has an interoperable binding, and the reference and service are in the same SCA domain.

In this section we've described a number of options for how SCA references and services can be connected, and it's worth summarizing these briefly:

- To connect a reference and service in the same SCA domain, you can do any of the following:
  - Use a wire with the default binding
  - Use a wire with an interoperable binding
  - Configure the reference and service with matching bindings
- To connect an SCA reference and SCA service that aren't in the same domain, you can configure the reference and service with matching interoperable bindings.
- To connect an SCA reference or service to non-SCA code, you can configure the reference or service with an interoperable binding.

Interoperable bindings enable SCA code to interoperate with non-SCA code and allow SCA code from different vendors to interoperate across a domain boundary. The default binding allows vendors to provide optimized communication where interoperability isn't required.

2.6 Summary

We've shown how SCA components can have implementations of various different types. The component definition configures selected aspects of the implementation, known as its component type. The component type includes services, references, and properties. Component references and services are connected using wires, or they can be configured using bindings to specify a particular communication protocol.

These concepts provide the essential foundations for SCA that you'll use when building applications with Tuscany. In a building, the foundations are essential but

usually not exciting. The exciting part is what's built on top of the foundations. In the next chapter, we'll move beyond the foundations and show how to add walls, doors, windows, upper stories, and a roof! To put this in SCA terms, we'll go on to explain how components can be composed and packaged into higher-level SCA constructs such as composites and contributions and deployed into an SCA domain using the Tuscany runtime in a local or distributed environment. With this additional information as well as the foundational concepts we've looked at in this chapter, you'll have a complete overview of everything involved in building an SCA composite application. Turn the page and come with us as we continue the journey.

3 SCA composite applications

This chapter covers

- Running a composite application in a single process
- Understanding the SCA domain
- Running a distributed composite application
- Using SCA composites as application building blocks

In this chapter we'll look at how to use SCA and Tuscany to create and run composite applications. As the name suggests, a composite application is an application composed from other things. These other things are the basic SCA artifacts that we looked at in the previous chapter: components, implementations, services, interfaces, references, wires, properties, and bindings. To combine these into a composite application, you'll need to understand the additional concepts of contributions, composites, domains, and execution nodes, and we'll take a detailed look at all of these in this chapter using the travel-booking application as an example.

The code samples in this chapter are based on the simplified travel-booking application that we introduced in chapter 1. In some cases the chapter 1 code is used unchanged, and in other cases it's extended to illustrate specific points. The

code in this chapter isn't used directly in the complete travel application. You can find the sample code for this chapter in the travel sample's directories contributions/introducing-tours, contributions/introducing-trips, contributions/introducing-client, and contributions/buildingblocks.

Composite applications are important because any realistic SCA application would contain more than a single component offering a single service. Also, the only thing that you can run in an SCA runtime, such as Tuscany, is an SCA composite. You can run a single component in Tuscany but only by creating and running a composite application that contains a single component. This might sound like an unnecessary complication, but it makes more sense when you consider that SCA is designed to represent applications with more than one component and could be used for applications with as many as hundreds of components. The power of SCA is that the composite application provides a precise and concise mechanism for describing and deploying the assembly of components.

We'll start by looking at how you can use Tuscany to create and run a composite application within a single process. We'll go on to explore the SCA domain and show how you can use it to create applications that can be distributed across multiple processes or a network of computers. Finally, we'll look at how you can make your composite applications easier to change and extend by using SCA composites to organize your components and services into modular assemblies that are easy to compose and recompose when business needs change.

3.1 Running a composite application in a single process

In the previous chapter you saw how to create composites containing your component definitions. The next step is to fire up the application and see how it runs! In this section we'll show you how to run a composite application within a single process and JVM (known as *local execution* in SCA terminology). Later in this chapter you'll see how you can use the SCA domain to run the same application in a distributed execution environment using multiple processes or a network of computers.

There are many ways to run a composite application with Tuscany. For example, the application can be run as a standalone Java main program or within a hosting environment such as a web application server. Chapter 11 gives details of all the possibilities. In this chapter we'll keep things simple by showing how to run a standalone Java application with Tuscany in local and distributed modes.

To run a local composite application in Tuscany, we'll need some contributions and some launcher code. We'll use the contributions and launcher code from the travel-booking application that we used in chapter 1. Figure 3.1 shows the structure of this sample application.

You can see that the composite application consists of three composites: client.composite, tours.composite, and trips.composite. Each of these composites contains one or more components, and there are wires connecting the component references to services in other components.

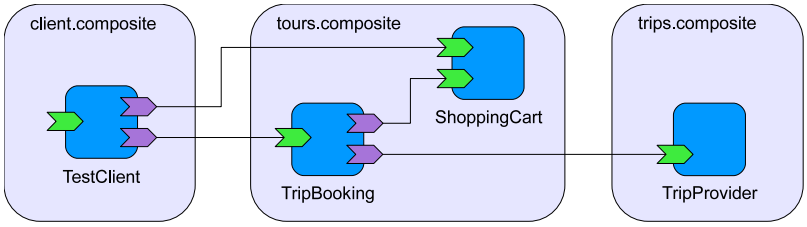


Figure 3.1 The sample composite application consists of three composites and four components, with wires connecting the component references to services in other components.

3.1.1 Preparing the contributions

The first step in running a composite application is to create contributions containing the code that you want to run. A contribution is an SCA packaging mechanism that can contain executable code, composites, and any other artifacts used by the application such as schema files or WSDL files. In Tuscany, a contribution can be a JAR or zip archive file, a directory structure, or a web application archive (WAR file). Details of how to use WAR files as contributions can be found in chapter 11.

To run the introductory travel-booking application that we'll use to illustrate this section, you need to install the Tuscany binary distribution and the Tuscany travel sample on your computer, and you'll need to build the travel sample. You can find full instructions for doing this in appendix A. After you've built the travel sample, you can find the code we're using for this section as the contents of the following directories relative to the top-level travelsample directory:

- A contribution in the directory contributions/introducing-tours/target/classes containing tours.composite and its implementation code
- A contribution in the directory contributions/introducing-trips/target/classes containing trips.composite and its implementation code
- A contribution in the directory contributions/introducing-client/target/classes containing test client code for components in tours.composite and trips.composite

Let's look inside the first of these contributions. It contains the following files:

- A number of .class files for Java implementation class and interface code in directories corresponding to their Java packages. For example, the .class file for the interface `com.tuscanyscatours.Bookings` is in the com/tuscanyscatours directory.
- An SCA composite definition in the tours.composite file in the root directory. Composite definitions need to have a .composite file extension; they can be anywhere in the contribution, and they can have any filename preceding the .composite extension.
- A META-INF directory containing a file named sca-contribution.xml. This file consists of a `<contribution>` element with child elements declaring the contribution's exports, imports, and deployable composites.

This contribution doesn't contain any other files. If other files (such as WSDL or XSD files) are needed, they can be placed anywhere in the contribution's directory structure.

Contributions can export and import XML namespaces and Java packages. This allows files to be shared between contributions and avoids the need to duplicate the same files in different contributions. Exports and imports are declared in the sca-contribution.xml file. The following example shows the sca-contribution.xml file from the `introducing-tours` contribution.

Listing 3.1 The sca-contribution.xml file from the introducing-tours contribution

```
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
              xmlns:tst="http://tuscanyscatours.com/">
    <export.java package="com.tuscanyscatours" />
    <deployable composite="tst:Tours" />
</contribution>
```

The `<export.java>` element exports the Java package `com.tuscanyscatours` from this contribution. It's also possible to export an XML namespace using the `<export>` element.

To import an exported Java package into another contribution, the importing contribution's sca-contribution.xml file needs to include an `<import.java>` element. You can see an example of this in the sca-contribution.xml file from the `introducing-client` contribution shown here.

Listing 3.2 The sca-contribution.xml file from the introducing-client contribution

```
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
              xmlns:client="http://client.scatours/">
    <import.java package="com.tuscanyscatours" />
    <deployable composite="client:Client" />
</contribution>
```

The `<import.java>` element declares that the `com.tuscanyscatours` package is imported into this contribution. To import an XML namespace, the `<import>` element would be used.

We've imported the `com.tuscanyscatours` package so that reference declarations in the `introducing-client` contribution can use the `com.tuscanyscatours.Bookings` and `com.tuscanyscatours.Checkout` interfaces from the `introducing-tours` contribution. Importing these interfaces means that we don't need to package separate copies in the `introducing-client` contribution.

As well as declaring exports and imports, the sca-contribution.xml file lists the deployable composites within the contribution. Each deployable composite is declared using a `<deployable>` element with a `composite` attribute specifying the XML qualified name of the composite. You can see in listings 3.1 and 3.2 that the `introducing-tours` and `introducing-client` contributions each contain a single deployable composite. You'll see later in this section how deployable composites are used.

We've shown how to prepare contributions for the introductory travel-booking application code. Next we'll look at how to write a launcher for the application.

3.1.2 Writing the launcher

A launcher is a small piece of code that uses Tuscany APIs to load contributions and run them using a Tuscany execution node. The launcher might also invoke services in the contributions it has loaded. The launcher code is customized for the contributions it loads and the services it invokes. Our launcher needs to load the following contributions:

- `contributions/introducing-tours/target/classes`
- `contributions/introducing-trips/target/classes`
- `contributions/introducing-client/target/classes`

Listing 3.3 shows the launcher code that we'll use to load these contributions and call a method `run()` of a service in the test client contribution. You first saw this code in chapter 1, and we'll describe what it does in more detail here. The code is in the travel sample directory launchers/introducing.

Listing 3.3 Launcher code to load an application's contributions and call a test method

```
package scatours;

import static scatours.launcher.LauncherUtil.locate;
import org.apache.tuscany.sca.node.SCAClient;
import org.apache.tuscany.sca.node.SCANode;
import org.apache.tuscany.sca.node.SCANodeFactory;

public class IntroducingLauncher {

    public static void main(String[] args)
           throws Exception {
        SCANode node = SCANodeFactory.newInstance()
              .createSCANode(null,
                  locate("introducing-tours"),
                  locate("introducing-trips"),
                  locate("introducing-client"));
        node.start();

        Runnable proxy = ((SCAClient)node)
              .getService(Runnable.class,
                   "TestClient/Runnable");
        proxy.run();

        node.stop();
    }
}
```

1 Find and create contributions

2 Get proxy for Runnable service

3 Call proxy's run() method

In listing 3.3 the `createSCANode()` call creates a local (in-process) execution node for the three contributions passed as arguments, which are returned by the three `locate()` calls 1. The first argument for the `createSCANode()` call is the name of the composite to be deployed. It's `null` in this case, which means the node's execution environment will be constructed by combining all the deployable composites in the contributions.

The execution node is returned as an `SCANode` object, and the launcher starts this node by calling its `start()` method. Starting the node creates executable runtime components from the component definitions in the deployable composites. The services in these components are now available for invocation, but there's no code running yet. To run some code, the launcher uses the `SCAClient.getService()` call ❷ to create a proxy for the Runnable service of the TestClient component and calls the proxy's `run()` method ❸, which in turn calls the `run()` method in the TestClient component. When this method returns, the launcher stops the node by calling its `stop()` method.

> **WARNING** If you pass multiple contributions to the `createSCANode()` method, you need to make sure that any contribution with imports is placed later in the sequence of contributions than the contribution that provides the corresponding exports. This is because of a limitation in the processing of imports and exports in the Tuscany 1.x runtime. In this example the `introducing-client` contribution imports a Java package from the `introducing-tours` contribution, so `introducing-tours` needs to appear before `introducing-client`.

You've seen what a launcher does and how to write it. In some simple cases it's possible to load contributions and create a node without writing any launcher code. See chapter 11 for details of how this works.

3.1.3 Running the launcher

The launcher is just a `main()` method of a Java class, so it can be run directly from the command line in the usual way. To run the launcher for the sample application, open a command prompt and navigate to the launchers/introducing directory. This directory contains the launcher code from listing 3.3 together with a suitable build.xml file. To run this launcher, enter the command

```
ant run
```

This uses the build.xml file in the launchers/introducing directory to run the launcher.

You've seen how to use Tuscany to run an application with three contributions in a single process. In the next section you'll see how the SCA domain supports distributed execution across multiple processes or a network of computers.

3.2 Understanding the SCA domain

The SCA domain contains all the executable code and other related application artifacts used by composite applications. It also includes the execution environment for SCA components and services. In this section we'll explore how the domain works, and you'll see how Tuscany makes the domain configuration available to a single execution node or a distributed network of execution nodes. The SCA domain is used for a number of purposes:

- It acts as a repository for the contributions used by the SCA runtime.
- It's the naming and visibility boundary for a distributed SCA runtime.
- It includes a local or distributed execution environment for SCA components.

In this section we'll explore these different facets of the domain. We'll start by looking at how SCA uses contributions to install application code and other related artifacts into the domain. Then we'll look at how SCA resolves references to named artifacts within the context of the domain. We'll show how composites are deployed into the domain to make their contents available for execution, and we'll look at different options for how to use one or more execution nodes within a domain. Finally, we'll show how Tuscany makes the domain configuration available to execution nodes.

3.2.1 The domain as a contribution repository

The executable code, composite definitions, and any other artifacts needed by a composite application are packaged into one or more SCA contributions. Before these contributions can be used by the SCA runtime, they need to be installed into the SCA domain. The mechanism used for installing contributions depends on the SCA runtime; the domain could be implemented as a repository, a registry, a filesystem, or in any other way. Tuscany installs a contribution into the domain by registering the contribution's URL and doesn't do any physical copying.

Installing a contribution into the domain code repository doesn't automatically make its contents available to the SCA runtime for execution. This requires the further step of deploying composites contained in the installed contributions, as described in section 3.2.3.

Contributions can share selected parts of their contents with other contributions by using import and export declarations for XML namespaces, Java packages, or other artifact types. Figure 3.2 illustrates how artifacts can be packaged in contributions and shared between contributions.

Here the `payment-bpel` contribution contains a composite payment.composite and some WSDL files that define types used by this composite. The `payment-bpel-process` contribution contains the file payment.bpel, which defines a BPEL process named Payment in the XML namespace http://tuscanyscatours.com/Payment and also contains some WSDL files that define types used by the BPEL process.

The XML namespace http://tuscanyscatours.com/Payment is exported by the `payment-bpel-process` contribution and imported by the `payment-bpel` contribution. This allows the `payment-bpel` contribution to access XML definitions in this

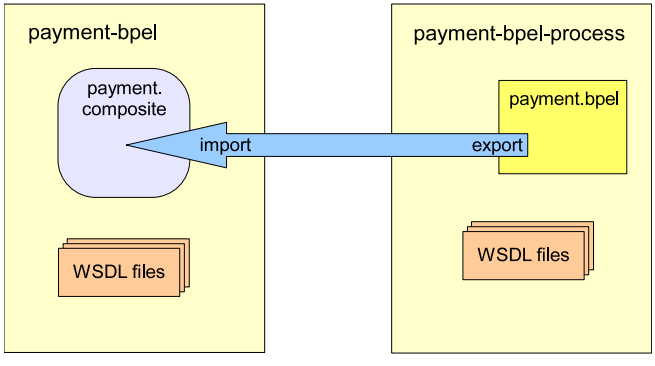


Figure 3.2 Exporting an XML namespace from the `payment-bpel-process` contribution and importing the same namespace into the `payment-bpel` contribution

namespace that are packaged in the payment-bpel-process contribution. In this case the BPEL process Payment that's defined in payment.bpel is referenced by the <implementation.bpel> element in payment.composite. There aren't any exports or imports for the XML namespaces used by the WSDL files, so the definitions in these files are visible only within their respective contributions.

It's possible to structure contributions and their exports and imports so that interface definitions used by a service are exported from the service's contribution and imported into other contributions that contain references wired to the service. This might seem attractive at first, but it's likely to cause problems because of the tight coupling it creates between the reference and the service. For example, a service with a Java interface might be changed to a different implementation with a compatible service interface in a different Java package or even in a different interface definition language. SCA's flexible interface matching is designed to ensure the reference still works in these cases, but this would fail if the reference interface were imported from the service's contribution.

Another option is to package shared interfaces in a separate contribution and import them into all the contributions that need them. This is more flexible than packaging shared interfaces with services, but it can create complex dependency relationships between contributions. Packaging duplicate copies of shared interfaces in all the contributions that use them is the simplest and most flexible approach, but it has the disadvantage of requiring updates to these interfaces to be coordinated across contributions. You'll need to consider these trade-offs when you choose a packaging approach for your applications.

SCA doesn't specify how contributions are installed into the domain. In Tuscany there are two mechanisms for doing this, depending on whether the domain is local to a single execution node or distributed across execution nodes. For a local execution node created by calling createSCANode(), as IntroducingLauncher does, the node contains its own domain, and contributions are installed into this domain by passing them as arguments to createSCANode(). For a distributed domain, contributions are installed using the domain manager, as you'll see in section 3.3.2.

You've seen how the domain provides a repository for contributions that include executable code and other artifacts. Now we'll look at how the domain handles naming and visibility for the things it contains.

3.2.2 The domain as a naming and visibility boundary

The SCA mechanisms for resolving names and accessing artifacts operate only within the scope of a single SCA domain. For example, contribution exports and imports can be resolved only within the same domain. This means that if a contribution named ShoppingCart exports the Java package com.tuscanyscatours.shoppingcart, the classes in this package within the ShoppingCart contribution can't be imported by an SCA contribution installed into some other domain, because exports and imports don't work across domain boundaries.

In the same way, SCA components can only be wired within a domain. If a component named TripBooking is deployed into an SCA domain and a reference in this component specifies a target service name of TripProvider/Trips, the reference and service can be wired only if the TripProvider component is deployed into the same SCA domain. If the TripProvider component is deployed into some other SCA domain, TripBooking can't be wired to TripProvider.

The naming and visibility boundary provided by the SCA domain doesn't prevent composite applications from using resources that aren't part of the domain. For example, an SCA component implementation can load Java classes using non-SCA mechanisms such as the Java class path, Java class loaders, or OSGi bundles. Classes loaded in these ways don't need to be packaged in an SCA contribution or be present in the domain. Also, if an SCA component needs to invoke a service that isn't in the domain, this isn't a problem; either the component can use a suitable SCA binding to invoke the service through an SCA reference, or the component's implementation code can use non-SCA APIs such as Web Services, JMS, or CORBA client APIs to make the invocation. This flexibility makes it easy for new code written using SCA to integrate with existing code that uses other approaches for locating resources and invoking services.

The domain boundary for SCA name resolution helps prevent accidental naming conflicts. For example, you might choose the name TripProvider for a component, and someone else might choose the same name for a different component. SCA component names don't use XML namespaces, so what's to stop these two component names from colliding with each other? The answer is the domain boundary. As long as the composites containing the duplicate component names are never deployed into the same domain, there's no chance of a name collision or of using the wrong component by mistake.

We've shown how naming and visibility work in the SCA domain. Next we'll look at how the SCA domain provides an execution environment (either local or distributed) for running components in the domain either with a single execution node or using a network of communicating execution nodes.

3.2.3 The domain as an execution environment

The execution context provided by the domain is represented as a special composite known as the domain composite. Unlike other composites that are defined statically in .composite files, the contents of the domain composite are maintained by the domain and are updated dynamically as a result of deployment actions.

When a composite is deployed to the domain, all of its contents become part of the domain composite, just as if the deployed composite had been included in the domain composite using an `<include>` element. Composite inclusion is described later in this chapter in section 3.4.3. Deployment makes the components in the deployed composite available for execution by the SCA runtime and also makes the services and references of these components available for wiring to other references and services in the domain.

SCA doesn't define how composites are deployed into the domain. In a local Tuscany execution node, the `createSCANode()` call deploys one or more composites, as discussed in section 2.1.2. For a distributed domain, composites are deployed using the domain manager, as we'll describe in section 3.3.3.

As an example, suppose that the TripBooking and TripProvider components are defined as follows:

```
<component name="TripProvider">
    <implementation.java
          class="com.goodvaluetrips.impl.TripProviderImpl" />
    <service name="Trips" />
</component>

<component name="TripBooking">
    <implementation.java
          class="com.tuscanyscatours.impl.TripBookingImpl" />
    <service name="Bookings" />
    <reference name="mytrips" target="TripProvider/Trips" />
</component>
```

When the TripProvider component is deployed to the domain, the Trips service of TripProvider becomes available for wiring. If the TripBooking component is then deployed to the domain, the Tuscany runtime wires the `mytrips` reference of TripBooking to the Trips service of TripProvider, and the Bookings service of TripBooking is available for wiring.

How big is a domain?

An SCA domain can be as big or as small as you want it to be. The domain's execution context can be shared by multiple processes on a computer or by multiple computers across a network. At the other end of the spectrum, Tuscany supports running one or more domains entirely within a single process and JVM. This flexibility in the scope of a domain enables SCA and Tuscany to meet a variety of application deployment needs ranging from small embedded systems all the way up to distributed enterprise environments.

Composites deployed into the domain are executed by the Tuscany runtime in one of the following ways:

- *Local domain*—The entire contents of the domain composite are used exclusively by a single execution node, with the node and domain running together locally within a single process. Other execution nodes belonging to different domains can also be running in the same process and JVM. You saw how to run Tuscany as a local domain in section 3.1.
- *Distributed domain*—Each composite deployed into the domain is assigned to its own execution node. This means that the components and services in the

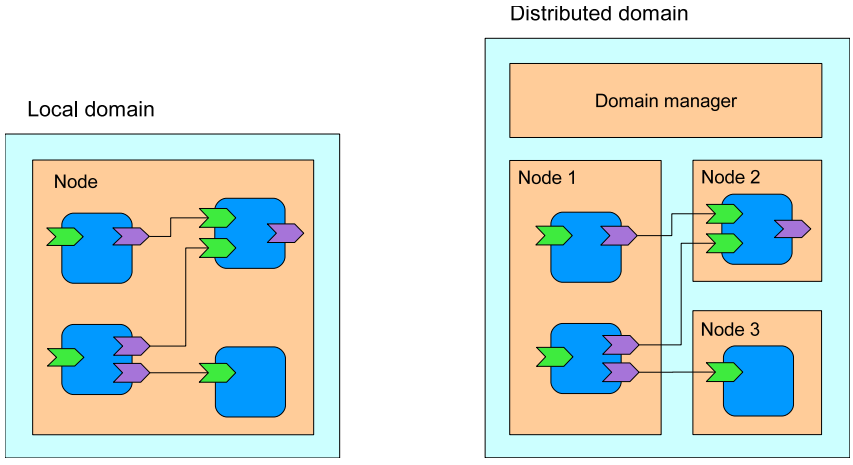


Figure 3.3 A local domain with a single execution node and a distributed domain with a domain manager and multiple execution nodes. The contents of the domain are the same in both cases.

domain are distributed across multiple execution nodes. Communication between the execution nodes is coordinated by the Tuscany domain manager. The execution nodes and domain manager can run in different processes on a single computer, or they can run on different computers across a network. The nodes use the domain manager at startup time to obtain network addressing information for services they provide and services they use. You'll see how to use the domain manager and how to run a Tuscany distributed domain in section 3.3.

Figure 3.3 illustrates these two kinds of domain configuration and shows how the same domain contents can be executed in either a local or distributed manner. In this figure the local domain contains a single execution node that runs the entire contents of the domain, and the components communicate using direct calls within a single JVM. The distributed domain uses the domain manager to allocate the components in the domain to three execution nodes, with remote network invocations between these nodes. The component implementations and component definitions are the same in both cases.

In the rest of this section we'll look at both of these approaches in more detail and see how they affect the ways that services are invoked and configured.

3.2.4 Using a single execution node with a local domain

A common case is for the entire SCA domain to run within a single execution node as a local domain. We showed how to do this in section 3.1. Because all the components in the domain are running within a single process and JVM, the Tuscany runtime automatically uses optimized direct calls for references and services that are wired using

binding.sca. For other bindings that communicate using network endpoints (such as binding.ws), the binding elements in the deployed composite need to specify the necessary network endpoint information. The Tuscany domain manager isn't used in this scenario.

As an example, let's look at three components: TripBooking, ShoppingCart, and TripProvider. We'll also be referring to these components later when we talk about distributed domains. The deployed composites containing these component definitions are shown here.

Listing 3.4 Deployed composites containing component definitions

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="Tours">
    <component name="TripBooking">
        <implementation.java
              class="com.tuscanyscatours.impl.TripBookingImpl" />
        <service name="Bookings" />
        <reference name="mytrips" target="TripProvider/Trips" />
        <reference name="cart" target="ShoppingCart/Updates" />
    </component>

    <component name="ShoppingCart">
        <implementation.java
              class="com.tuscanyscatours.impl.ShoppingCartImpl" />
        <service name="Checkout" />
        <service name="Updates" />
    </component>
</composite>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://goodvaluetrips.com/"
           name="Trips">
    <component name="TripProvider">
        <implementation.java
              class="com.goodvaluetrips.impl.TripProviderImpl" />
        <service name="Trips" />
    </component>
</composite>
```

The references in TripBooking are wired to services in TripProvider and ShoppingCart using binding.sca. If all these components are running in the same execution node and TripBooking calls TripProvider using the mytrips reference, the Tuscany runtime will invoke the TripProvider code by a direct call within the same JVM. The same thing will happen when TripBooking calls ShoppingCart using the cart reference.

You've seen how running a local domain allows Tuscany to make direct calls for service invocations across wires that use binding.sca. Next you'll see how the same service invocations are handled when using a distributed domain with the Tuscany domain manager.

3.2.5 Distributed execution within a domain

SCA is designed to support distributed execution across multiple processes or computers within the same SCA domain. In Tuscany, this setup is represented by a number of execution nodes and a separate domain manager. The network endpoints of all the execution nodes are registered with the domain manager, and the nodes use the domain manager to obtain network endpoint information for their services and references. The details of how to register network endpoint information with the domain manager are provided later in this chapter in section 3.3.4.

It's important to point out that the Tuscany domain manager isn't a distributed runtime for executing SCA components. These components are executed by the nodes to which they have been assigned, and the nodes execute without any interaction with the domain after their initial setup. Also, the domain manager doesn't make decisions on which components should be executed by which nodes. The domain manager acts as a registry that holds information about execution node assignments and endpoint configurations and provides this information to execution nodes when requested.

As an example, we'll take the same three components—TripBooking, ShoppingCart, and TripProvider—that we used in listing 3.4. When running these components in a distributed configuration using the domain manager, the same deployed composites and component definitions are used. These component definitions don't say anything about the execution-time locations of TripBooking, ShoppingCart, and TripProvider.

In this example let's suppose that we want to run the deployed composite Tours containing the TripBooking and ShoppingCart components on a computer with hostname `tuscanyscatours.com` and port number 8081, and run the deployed composite Trips containing the TripProvider component on another computer with hostname `goodvaluetrips.com` and port number 8082. Figure 3.4 illustrates this configuration.

With the distributed configuration shown in figure 3.4, the Tuscany runtime can't use optimized direct calls when TripBooking invokes TripProvider, because these components are running on different computers across a network. Instead, the runtime

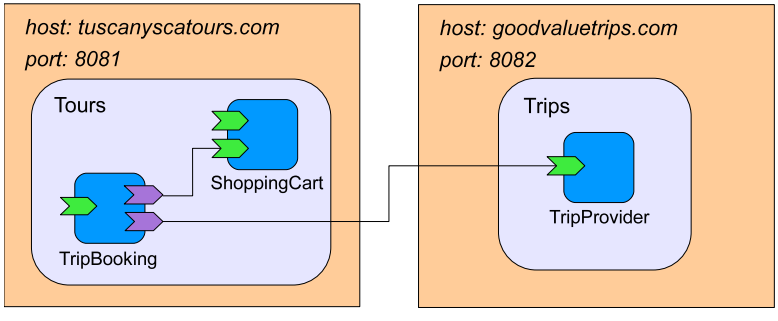


Figure 3.4 Running the deployed composites Tours and Trips on two computers with hostnames `tuscanyscatours.com` and `goodvaluetrips.com`

for TripBooking needs to make these calls by sending network requests to a remote endpoint that the runtime for TripProvider has exposed.

Tuscany handles this by having each execution node contact the domain manager to get a modified version of the node's deployed composite with resolved network endpoint information for the component services and references. The endpoint information for each node needs to have been registered previously with the domain manager, and we'll show how to do this in section 3.3.4. When the node's `start()` method is called, it contacts the domain manager to get the modified deployed composite for the node. For the execution node for `tuscanyscatours.com`, the component definitions in this modified composite are as follows:

```
<component name="TripBooking">
    <implementation.java
          class="com.tuscanyscatours.impl.TripBookingImpl" />
    <service name="Bookings">
        <binding.sca uri="http://tuscanyscatours.com:8081/TripBooking" />
    </service>
    <reference name="mytrips">
        <binding.sca uri="http://goodvaluetrips.com:8082/TripProvider" />
    </reference>
    <reference name="cart">
        <binding.sca uri="/ShoppingCart/Updates" />
    </reference>
</component>

<component name="ShoppingCart">
    <implementation.java
          class="com.tuscanyscatours.impl.ShoppingCartImpl" />
    <service name="Checkout">
        <binding.sca uri=
            "http://tuscanyscatours.com:8081/ShoppingCart/Checkout" />
    </service>
    <service name="Updates">
        <binding.sca uri="/ShoppingCart/Updates" />
    </service>
</component>
```

These modified component definitions are the same as the component definitions in listing 3.4, with the addition of binding URIs for all the services and references. These URIs are either network endpoint URIs or local URIs, depending on whether remote or local invocations are required. In this example the Bookings service, the Checkout service, and the `mytrips` reference have network endpoint URIs because they need to use remote invocation, and the Updates service and the `cart` reference have local URIs because they're running in the same process and can use the optimized invocation path for `binding.sca`.

The other execution node running on `goodvaluetrips.com` gets the following modified component definition from the domain manager:

```
<component name="TripProvider">
    <implementation.java
```

```
          class="com.goodvaluetrips.impl.TripProviderImpl" />
    <service name="Trips">
        <binding.sca uri="http://goodvaluetrips.com:8082/TripProvider" />
    </service>
</component>
```

The Trips service in this component has a network endpoint URI so that it can receive remote invocations from the TripBooking component.

The domain manager can also hold network endpoint configuration for binding types other than `binding.sca`, such as `binding.ws`. To use the domain manager with `binding.ws`, we could change the component definitions in the deployed composites by adding empty `<binding.ws/>` elements on the Bookings service, the `mytrips` reference, and the Trips service. We'd also need to add endpoint configuration for `binding.ws` to the domain manager registry. With these changes, the domain manager will add the correct network endpoint information to the `<binding.ws/>` elements in the modified component definitions that it sends to the execution nodes.

It's sometimes necessary to change the network endpoint of an execution node after it's been started and is running services. After the necessary changes have been made to the domain manager configuration, the affected node needs to be restarted so that it reloads its network endpoint information from the domain manager. It's also necessary to stop and restart all other nodes that use services running in the restarted node, so that the endpoint information in these other nodes for references to the updated node can be refreshed.

You've seen what the SCA domain provides and the different ways it can be used when running composite applications. A single execution node can act as an SCA domain, or the Tuscany domain manager can be used to control a domain involving many execution nodes. Where more than one execution node is running in the SCA domain, these execution nodes can be running in the same JVM, in different JVMs on the same computer, or on different computers. Regardless of the physical configuration of execution nodes, the SCA domain is doing the same job of providing a repository of contributions, defining the visibility boundary for contributed artifacts and SCA component names, and providing fully configured composite definitions to execution nodes.

In section 3.1 we showed how you can run a composite application with three contributions using a single process. In the next section we'll show how you can run the same three contributions in a distributed execution environment using the Tuscany domain manager.

3.3 Running a distributed composite application

To run a distributed composite application in Tuscany, a number of steps are needed. In this section we'll go through these steps in the order in which you need to do them, using the same sample application that we used in section 3.1.

Figure 3.5 shows this application running in a distributed configuration. This figure shows the same composites and components that you saw in figure 3.1. In the distributed configuration these composites and components have been allocated to

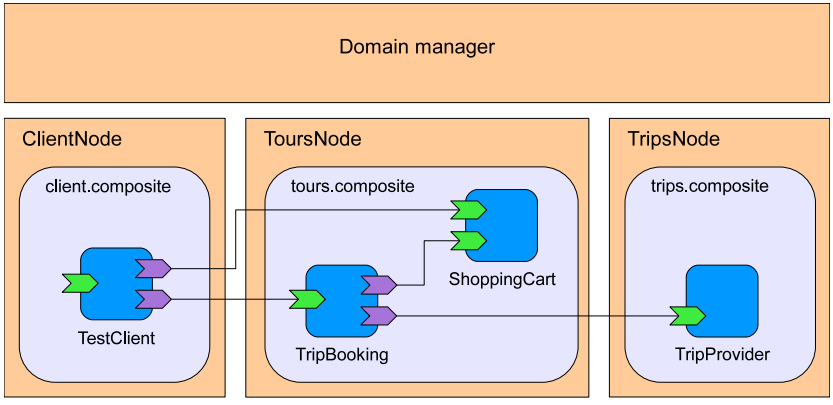


Figure 3.5 The sample application running in a distributed configuration, with three execution nodes coordinated by a separate domain manager

the execution nodes ClientNode, ToursNode, and TripsNode. These execution nodes are running as separate processes, and they're coordinated by the Tuscany domain manager.

To run this distributed application, we'll begin by creating an empty SCA domain using the Tuscany domain manager. Next we'll install the contributions for our application into the domain. Then we'll need to deploy the composites containing the executable components for the application. After this, we can define the location and binding information for the execution nodes that will run the deployed composites. The final step is to run the distributed application by creating execution nodes that load and run the deployed composites.

3.3.1 Creating an SCA domain

In Tuscany the state of the SCA domain is maintained by the domain manager, which usually runs as a separate process. This process can be started using the following command line:

```
java -cp <tuscany-dir>/lib/tuscany-sca-manifest.jar org.apache.tuscany.sca.
➥ node.launcher.DomainManagerLauncher
```

You must enter this command as a single line, replacing `<tuscany-dir>` with the installation directory for the Tuscany binary distribution. This is quite a long command to type, so the travel sample includes Ant scripts to do this where needed. It's also possible to use Tuscany APIs to run the domain manager within an existing process.

When the domain manager starts up, it looks to see if the current working directory contains a saved domain configuration. The domain configuration files are described in chapter 11. If there's a saved configuration, the domain manager uses this configuration for the initial state of the domain. If no configuration files are found, the domain manager creates an empty domain. In this example we're showing the complete process of building a domain from scratch, so you should start the

domain manager from a directory that doesn't contain a domain configuration. In the travel sample code tree there's a directory called testdomain, which is provided for this purpose, and in the rest of this section we'll assume you've started the domain manager from this directory. You can do this by opening a new command prompt, navigating to the testdomain directory, and typing `ant run` to start the domain manager.

If anything goes wrong while you're going through this example, you might want to delete your domain configuration and make a clean start. To do this, you can stop the domain manager by typing `q`, and then run either `ant clean` or `mvn clean` from the testdomain directory.

When the domain manager has loaded the startup configuration, it starts an HTTP listener on port 9990. This port is used to provide a browser-based user interface (the Domain Manager GUI) , which you can access at the URL http://<domainhost>:9990/ui/workspace where <domainhost> is the hostname of the computer running the domain manager.

> **WARNING** The Domain Manager GUI has compatibility issues with some browsers. It works in Firefox, Safari, and Chrome but has problems when running in Internet Explorer and Opera.

The domain manager also supports programmatic access to domain resources using HTTP GET and POST requests. If you want to use this advanced capability, you can find some information in the article "Deploy an SCA application using the Tuscany domain manager" listed on the Books and Articles page on the Tuscany website, and you can refer to the Tuscany source code for full details of how to use this protocol. All interactions with the domain manager (either from a browser or using the API) are by HTTP requests to its listener port.

3.3.2 Installing contributions into the domain

You can use the Domain Manager GUI to install new contributions into the domain and to manage installed contributions. To do this, navigate to the Contributions page by browsing to http://<domainhost>:9990/ui/workspace, where <domainhost> is the hostname of the computer running the domain manager, or click the Contributions link from any page in the Domain Manager GUI.

The Contributions page displays a list of the contributions already installed in the domain. If the domain manager is started from a directory without a saved domain configuration, this list will be empty. To install a new contribution into the domain, click the Add link of the Contributions page. This brings up an Add Contribution section with entry fields to enter the contribution URI and the contribution location.

To run the travel-booking application you need to install three contributions. Let's start with the `introducing-tours` contribution. The contribution URI is an identifier of your choice that will uniquely identify the installed contribution within the domain. To keep life simple, let's use `introducing-tours` as the URI.

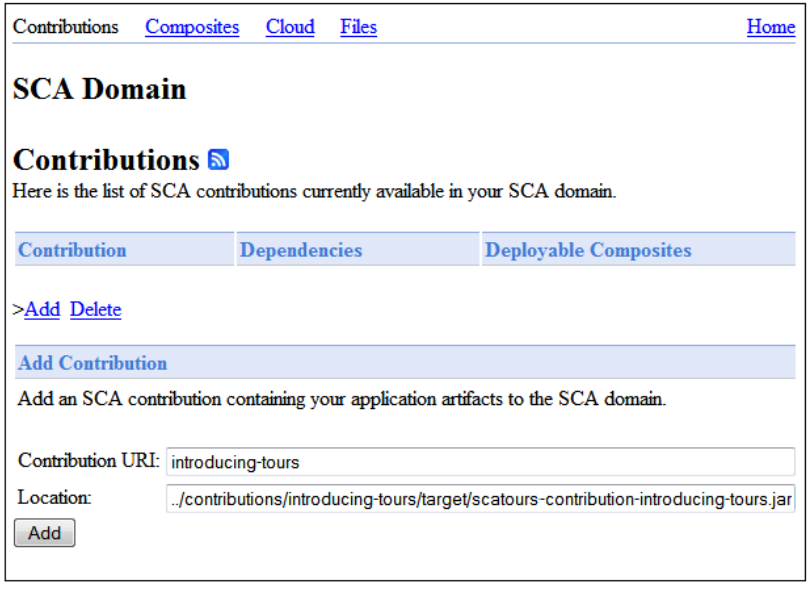


Figure 3.6 Adding a contribution to the domain using the Contributions page of the Domain Manager GUI

For the contribution location you can use a URL (for example, an http: URL), or you can use a file path if the domain manager and the execution nodes are running as different processes on the same computer. The file path can be absolute, or it can be a relative path from the directory that you used to launch the Domain Manager GUI. The contribution URL or file path can point to a JAR or zip file or to a directory.

If you started the domain manager from the testdomain directory, you can use the location file path ../contributions/introducing-tours/target/scatours-contribution-introducing-tours.jar. This relative file path works because you're going to run the execution nodes as different processes on the same computer. Figure 3.6 shows the result of entering this URI and location in the Add Contribution section.

After entering the URI and location, click the Add button to install the contribution into the domain. For the second contribution, you can use the URI `introducing-trips` and path ../contributions/introducing-trips/target/scatours-contribution-introducing-trips.jar. The final contribution is the test client, with the URI `introducing-client` and the path ../contributions/introducing-client/target/scatours-contribution-introducing-client.jar. When you've finished adding contributions, you can hide the Add Contribution section of the page by clicking the Add link (not the Add button).

Figure 3.7 shows how the Contributions page will look after installing all three contributions for the introductory travel-booking sample application. The Contribution column lists the URIs of all the installed contributions. The Deployable Composites column lists the XML qualified names of the deployable composites in each contribution, with the namespace and local name separated by a semicolon. The

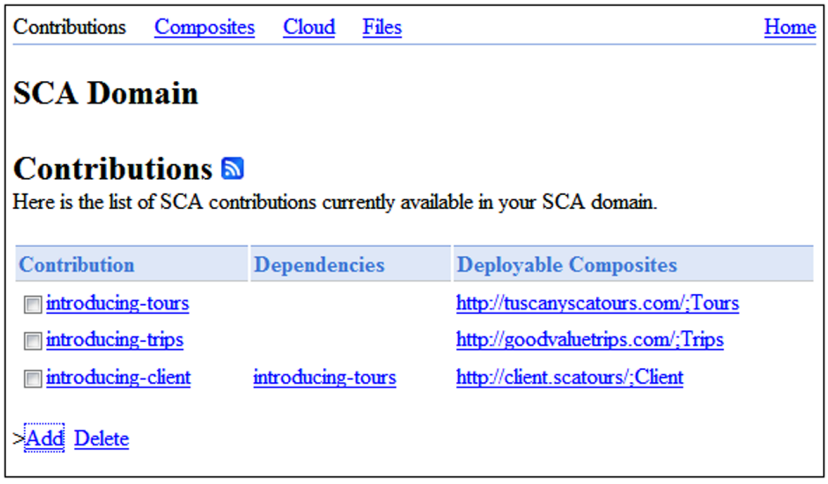


Figure 3.7 The Contributions page of the Domain Manager GUI after all the contributions for the introductory travel-booking application have been installed in the domain

Dependencies column shows which contributions depend on other contributions, based on their imports and exports. In this example, the `introducing-client` contribution imports the Java package `com.tuscanyscatours`, which is exported by the `introducing-tours` contribution.

The domain manager checks the contribution dependency relationships and reports any errors found by displaying error messages in the Dependencies column. For example, a contribution dependency might not be found in the domain. This could be caused by an incorrect import or export declaration or because the contribution containing the export hasn't yet been installed in the domain.

3.3.3 Deploying composites for execution

Before any components can be executed in the domain, the composites containing the components need to be deployed. Deploying a composite adds it to the domain's distributed execution environment and makes it available to run on an execution node. To deploy a composite using the Domain Manager GUI, you'll need to display the Composites page of the Domain Manager GUI by clicking the Composites link from any other page or by browsing directly to http://<domainhost>:9990/ui/composite, where <domainhost> is the hostname of the computer running the domain manager.

The Composites page displays a list of the composites that have been deployed for execution. Installing a contribution doesn't automatically deploy any composites within the contribution, so this list will be empty at first. To deploy a composite, click the Add link on the Composites page to reveal an Add Composite section containing entry fields for the composite namespace, the composite name, and the URI of the contribution containing the composite. There's no need to type anything into these fields, because you can click inside each field to see a selection prompt with all the

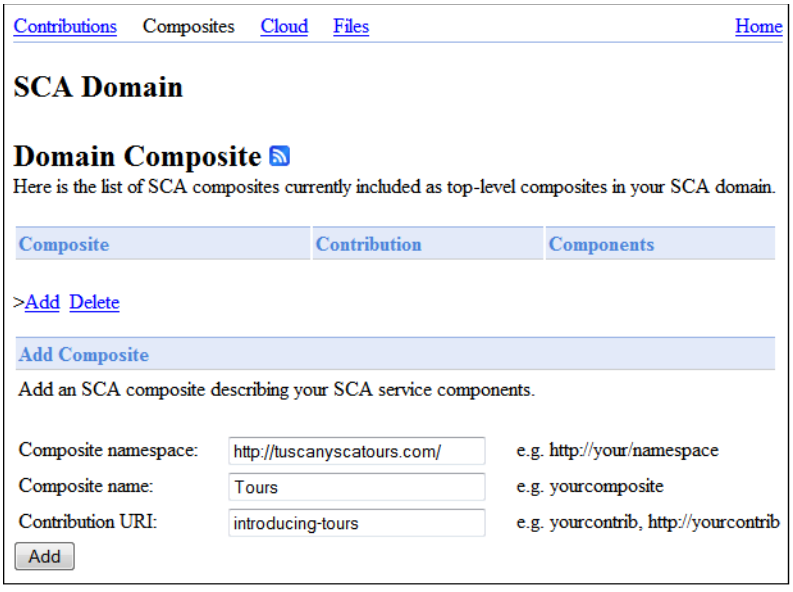


Figure 3.8 The Composites page of the Domain Manager GUI showing the contents of the entry fields for deploying a composite to the domain

possible choices for the deployable composites that are installed. Figure 3.8 shows the values you need to enter to deploy the Tours composite.

Note that composites are identified by their XML names (namespace and local part), not by their filenames. After clicking the Add button to deploy the Tours composite, you can repeat the same procedure to deploy the Trips and Client composites. After clicking the Add link to remove the Add Composite section, the Composites page should look like figure 3.9, with the qualified XML name of each deployed

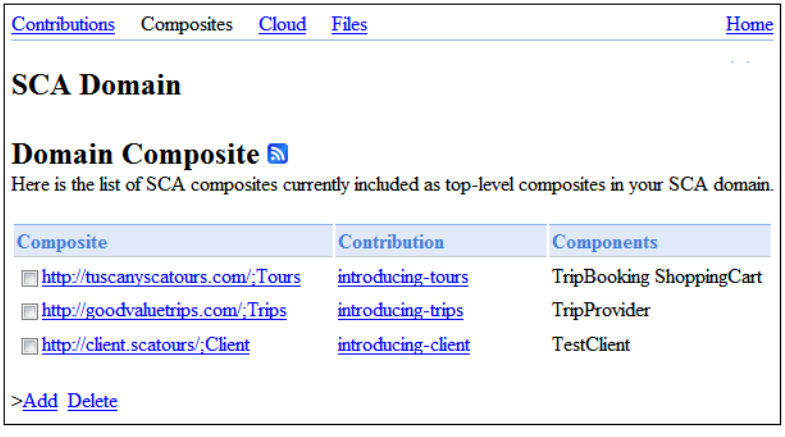


Figure 3.9 The Composites page of the Domain Manager GUI after all the composites in the application have been deployed

composite, the URI of the contribution containing the composite, and the names of the components in the composite.

Now we're ready to assign the deployed composites to execution nodes that will run the executable code for the component implementations in the composites.

3.3.4 Assigning composites to execution nodes

The final step before running our distributed application is to tell the domain manager about the execution nodes that will run the deployed composites. To do this, you need to bring up the Cloud page of the Domain Manager GUI by clicking the Cloud link at the top of any other page or entering the URL http://<domainhost>:9990/ui/cloud, where <domainhost> is the hostname of the computer running the domain manager.

To add an execution node to the domain configuration, click the Add link on the Cloud page. This displays an Add a Node section with fields for the node information and the composite that will run on the node, as shown in figure 3.10.

The node name can be any name that you choose to identify this execution node. The node URI contains the protocol, hostname, and port number that the node's runtime will use to listen for service requests. In this example we'll use `http:` as the protocol, `localhost` as the hostname (the nodes will run as separate processes on the same computer), and 8081, 8082, and 8083 as the port numbers for the Tours, Trips, and Client composites respectively. The Composite namespace, Composite name,

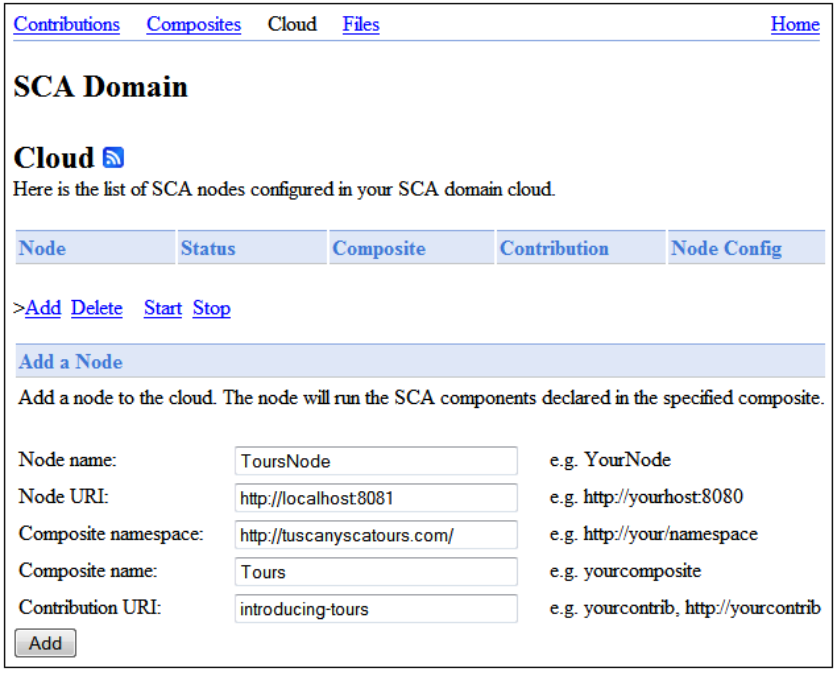


Figure 3.10 The Cloud page of the Domain Manager GUI showing an example of what would be entered to add an execution node for a deployed composite

and Contribution URI fields identify the deployed composite that will be run by this node. Instead of typing into these fields, you can click inside them to get selection prompts for all deployed composites.

Node with multiple protocols or listener ports

The Add a Node section of the Cloud page assumes that the node uses the same protocol, hostname, and port number for all the services offered by the deployed composite. For composites with a number of different types of service bindings, this limitation may be a problem. In such cases you can edit the domain manager configuration files directly to specify different listener protocols or ports for different service bindings, as described in chapter 11.

Clicking the Add button adds the ToursNode execution node to the domain configuration. In the same way you can define the TripsNode and ClientNode execution nodes for the Trips and Client composites. After you click the Add link to remove the Add a Node section, the Cloud page should look like figure 3.11.

This figure shows the name of each node, the name of the deployed composite that will run on the node, and the URI of the contribution containing the composite. All the nodes are shown as stopped in the Status column because the domain manager hasn't been told to start them. The Node Config column contains symbols representing Atom feeds that are used by each node's runtime to get its startup configuration. If you click this symbol and then click the first entry in the feed, you'll see the domain manager's version of the deployed composite (with fully resolved absolute binding URIs) that will be sent to the execution node.

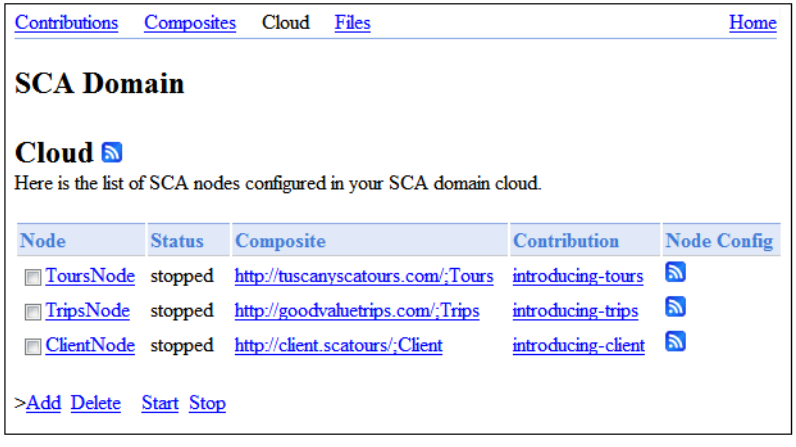


Figure 3.11 The Cloud page of the Domain Manager GUI after defining all the execution nodes

The Cloud page has a Start link that can be used to start execution nodes within the domain manager process. Don't try this at home! It was added to the domain manager user interface as a testing convenience; however, execution nodes are best run in some other process so that any failure doesn't bring down the domain manager or compromise its integrity. If you don't use the Start link, the Status column will always show the nodes as stopped, even if they've been started in other processes. We're ready to do this now.

3.3.5 Creating and starting execution nodes

The components within an SCA domain run on one or more execution nodes. The SCA specifications leave it up to implementations to define APIs for creating execution nodes and giving them components to run. In Tuscany these capabilities are provided by the `SCANodeFactory` class. Tuscany also provides a standalone node launcher that can be run from the command line. You can run the node launcher by entering the following command:

```
java -cp <tuscany-dir>/lib/tuscany-sca-manifest.jar org.apache.tuscany.sca.
➥ node.launcher.NodeLauncher http://<domainhost>:9990/node-config/<node>
```

You must enter this command as a single line, replacing `<tuscany-dir>` with your installation directory for the Tuscany binary distribution. You also need to replace `<domainhost>` with the hostname of the computer running the domain manager and replace `<node>` with the node name you used when creating the node in the Domain Manager GUI (for example, ToursNode). To avoid the need to type all this, the travel sample includes Ant scripts for running this command where it's needed (that is, in the launchers/introducing-tours and launchers/introducing-trips directories).

The http: URI argument passed to the `NodeLauncher` class is the URI of an Atom feed that the node launcher uses to obtain the node's configuration from the domain manager. For example, if the domain manager is running in a different process on the same computer as the node, the URI of the Atom feed for the ToursNode configuration would be

```
http://localhost:9990/node-config/ToursNode
```

You can open this as a URL in your browser if you're interested in what information is sent to the node by the domain manager. This can be useful to debug problems if the node doesn't start correctly for some reason.

The Tuscany node launcher reads the information from the Atom feed and uses this to configure and start the node, and then waits at a command prompt. The services running on the node are now available and ready to handle method invocations.

In this example, we have two nodes running services for our application: ToursNode and TripsNode. For convenience, the sample code has launcher directories for these nodes with Ant scripts to start the nodes. To run the launcher for ToursNode, you can open a new command prompt, navigate to the launchers/introducing-tours directory,

and type `ant run` to run the launcher command. The node startup should complete successfully and wait for input from the command line. The next step is to open another new command prompt and type `ant run` in the launchers/introducing-trips directory.

The application's services are now ready for action! To run them we'll need some client code that will invoke these services. It's good practice to use separate contributions for business service code and test client code, so we've created another contribution named `introducing-client` containing our test client code. To run this client code, we'll need to write a small launcher program, which uses the `SCANodeFactory` API to create a test client node and then invokes a service method on this node. The test node runs in the same process as the launcher code. The following listing shows this launcher program.

Listing 3.5 A simple launcher that creates a test client node and runs some test code

```
package scatours;

import org.apache.tuscany.sca.node.SCAClient;
import org.apache.tuscany.sca.node.SCANode;
import org.apache.tuscany.sca.node.SCANodeFactory;

public class IntroducingClientLauncher {

    public static void main(String[] args)
            throws Exception {
        SCANode node = SCANodeFactory.newInstance()
            .createSCANodeFromURL(
            "http://localhost:9990/node-config/ClientNode");
        node.start();

        Runnable proxy = ((SCAClient)node).getService(
                    Runnable.class, "TestClient/Runnable");
        proxy.run();

        node.stop();
    }
}
```

1 Create node to run test client code

2 Get proxy for Runnable service

3 Call service's run() method

The code in listing 3.5 is similar to the launcher code that you saw in listing 3.3. Here the launcher code creates a client node using the `createSCANodeFromURL()` method 1 of the `SCANodeFactory` class. The `SCANode` object returned by this call represents an execution node that will run in the same process as the launcher code. The launcher calls the node object's `start()` method to start the node. Next the launcher code uses the `SCAClient.getService()` method 2 to get a proxy for the Runnable service of the TestClient component and calls the proxy 3 to run the test code. When the test code returns, the launcher calls the node's `stop()` method to stop the node's services.

To run this test client launcher, open a new command prompt, navigate to the launcher/introducing-client directory, and type `ant run`. The launcher will run the

test client code on the ClientNode node in one process. This code will call services in the Tours composite running on the ToursNode node in another process, which will in turn call services in the Trips composite running on the TripsNode node in a third process.

In this example we've run three execution nodes and the domain manager in different processes on a single computer, as shown in figure 3.5. To run these on four different computers in a network, you'd only need to change the node URI settings in the domain manager's Cloud configuration and the configuration URIs passed to the nodes at startup. You wouldn't need to make any changes to the application contributions or to any other domain configuration settings. This ability to quickly and easily change the deployment configuration in a distributed runtime is a key feature of Tuscany and SCA and is made possible by SCA's clean separation between business service configuration and runtime deployment configuration.

3.3.6 Running the domain manager from a saved configuration

Congratulations! You've created an SCA domain from scratch using the Tuscany Domain Manager GUI, and you've run a distributed composite application within your domain. Now it's time to take a well-earned break from Tuscany and do something completely different for a while before coming back to continue your Tuscany domain adventures. When you fire up the domain manager again, you probably won't want to repeat all the same steps to set up your domain configuration from scratch using the Domain Manager GUI. The good news is that there's no need to repeat anything you did previously because the domain manager automatically saves the current domain configuration to disk whenever you do anything in the Domain Manager GUI, and it restores this saved configuration the next time it's started.

The domain configuration is saved as a number of XML files. By default, these files are stored in the working directory from which the domain manager was started. The next time you start the domain manager from the same directory, it will automatically find the previously saved configuration and use it to restore the domain to its previous state.

If you're configuring the domain using the Domain Manager GUI or if you're restoring a previously saved domain configuration, there's no need for you to understand the contents of the files that the domain manager uses to save the domain configuration. However, you might want to modify an existing domain configuration by directly editing these configuration files, or you might want to create a predefined domain configuration for the domain manager to use when it's started. To do these things you'll need to understand what the domain manager configuration files contain, and you'll find a full description of this in chapter 11.

We've looked at how you can run a composite application using either a local configuration or a distributed configuration. In both of these cases, our example application has been made up of components with self-contained implementations and SCA wiring to connect them together. For some applications, this simple, flat structure

with custom-built components is fine; in other cases, it's worth doing a bit of extra work to design parts of the application as reusable building blocks that can also be used in other applications. In the next section we'll show you how to create these reusable building blocks using SCA composites.

3.4 Using SCA composites as application building blocks

You've seen how SCA composites are used to contain component definitions for deployment into the SCA domain. SCA composites have another important use as building blocks in the internal structure of a composite application, as you'll see in this section.

Conventional applications have a hierarchical structure with a top-level layer of code that's built specifically for the application and isn't designed to be reused. Lower layers often use reusable libraries. In a composite application this hierarchical approach is replaced by a network of service connections between components. New components are created by combining existing components in different ways, and new services are created by wrapping and extending existing services. The resulting application is a loose federation of software components and services, organized to perform a particular task. These components and services are expected to be reused in ways that their original authors didn't anticipate, so they're designed to be as flexible and configurable as possible, and they make no assumptions about running in a specific environment.

An example of a composite application is the travel-booking application that we use in this book. Some components and services such as those that handle payments or manage the contents of a shopping cart aren't specific to travel booking and have been designed to avoid making travel-specific assumptions that would prevent reusing them for other purposes. Other services such as hotel reservation can either be consumed directly by a customer or used to compose other services such as booking a packaged trip with flights, hotel, rental car, and other activities. The result isn't a single specialized application in the traditional sense but is more like a collection of many different applications that can be used for a variety of purposes and extended further to meet additional needs.

To implement this kind of composite application we'll need some reusable building blocks that can be composed and recomposed in many different ways. In SCA, these building blocks are implemented as composites. We've already shown how SCA composites are used to define components that will be deployed into the domain. In this section we'll complete the picture by showing how to use composites to create building blocks for composite applications. We'll begin by looking at using composites as component implementations, and then we'll show how a composite can include other composites. By providing these different ways to use composites as building blocks within your applications, SCA gives you the flexibility to choose the best approach for each application.

3.4.1 Different ways of using SCA composites

SCA composites can be used as component implementations. When used in this way, a composite contains an assembly of components, which together form a higher-level component that acts as a black box for what's inside. For example, a computer storage drive can be manufactured from rotating platters (a hard disk drive) or from solid-state memory (a solid-state disk) together with other components such as a read/write cache and firmware. The drive is itself a component that can be used in higher-level assemblies such as a personal computer or network-attached storage. When plugging this drive into a computer, the other components in the system don't need to know whether the internal technology of the drive is hard disk or solid state, because this doesn't affect how other system components use the drive.

SCA composites can also be used by inclusion within other composites. When used in this way, a composite is a white-box container for the things it contains. For example, a self-assembly electronics kit for building an amplifier might contain transistors, resistors, capacitors, controls, and a display. Some kits are designed to provide choices between different design alternatives that require the kit components to be connected in different ways. The person assembling the kit needs to choose one of the alternatives and connect the components in the right way to produce the desired result.

Both of these approaches have their advantages. The black-box approach is more disciplined because the composite needs to be designed with a formal contract for how it can be used as a component implementation. The white-box approach is more flexible because there are no limitations on what the composite can contain, and the composite's internal structure is fully exposed.

When you create a composite, the composite definition doesn't say how the composite will be used. If the composite is designed with a formal contract, it can be used either as a component implementation or by inclusion in another composite. If the composite doesn't have a formal contract, it can be used only by inclusion in another composite. In the rest of this section we'll show you how to do both of these.

SCA composites are also used as containers for deploying components and wires. This is a special case of reuse by inclusion, where the deployed composites are included in the domain composite provided by the SCA runtime, as described in section 3.2.3.

3.4.2 Using composites as component implementations

You can use a composite as the implementation of a component. To do this, you need to define the services, references, and properties of the composite, which together make up the component type of the composite implementation. Figure 3.12 shows an example of this in diagrammatic form.

In this figure, some services, references, and properties are attached to the edges of the lighter outer box representing the Tours composite. These are known as composite services, composite references, and composite properties to distinguish them from the component services, component references, and component properties that

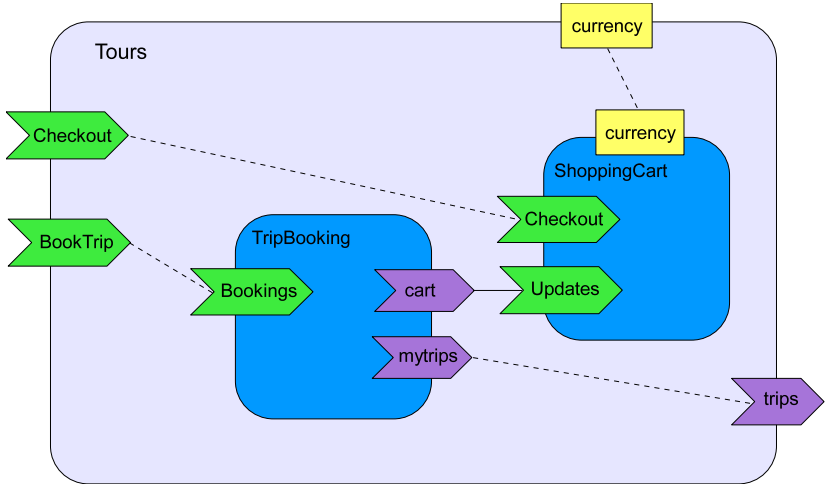


Figure 3.12 The Tours composite can be used as a component implementation, with composite services BookTrip and Checkout, a composite reference `trips`, and a composite property `currency`.

are attached to the darker boxes representing the TripBooking and ShoppingCart components. Composite services, references, and properties are visible outside the composite when the composite is used as a component implementation.

The composite services and references in figure 3.12 are shown connected to component services and references using dashed lines. These component services and references will be exposed outside the composite when the composite is used as a component implementation. Exposing a component service or reference in this way is described as *promoting* it.

An example of a promoted service is the Bookings service of the TripBooking component, which is promoted by the BookTrip composite service. This means that invoking the BookTrip service from outside the composite will actually invoke the Bookings service and execute code in the `com.tuscanyscatours.impl.TripBookingImpl` implementation class. Similarly, the `trips` composite reference promotes the `mytrips` component reference, which means that wiring a service to `trips` will have the effect of wiring it to `mytrips`. Any component services and references that aren't promoted aren't exposed by the composite implementation.

Composite services and references can have different names from their promoted component services and references, or they can have the same names. For example, the composite service for the promoted service Checkout is also called Checkout. There's no ambiguity because these names are used in different contexts.

Composite properties don't use promotion. Instead, component properties can be configured to get their values from composite properties. This relationship is shown

in figure 3.12 by the dashed line connecting the `currency` composite property to the `currency` component property.

A composite used as a component implementation needs to have a complete external contract. For example, it can't contain a component with a reference of multiplicity `1..1` or `1..n` that isn't either wired within the composite or promoted by the composite. An unpromoted reference isn't visible outside the composite, so it wouldn't be possible to satisfy the reference's multiplicity requirement by wiring it to a service.

The next listing shows how to write an XML definition for the Tours composite pictured in figure 3.12. You can find this composite in the travel sample's contributions/buildingblocks directory as the file tours.composite.

Listing 3.6 Creating a composite for use as a component implementation

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
      targetNamespace="http://bb.tuscanyscatours.com/"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      name="Tours">

    <service name="BookTrip"
          promote="TripBooking/Bookings" />
    <service name="Checkout"
          promote="ShoppingCart/Checkout" />
    <reference name="trips"
          promote="TripBooking/mytrips" />
    <property name="currency"
          type="xs:string">USD</property>

    <component name="TripBooking">
        <implementation.java class=
              "com.tuscanyscatours.impl.TripBookingImpl" />
        <service name="Bookings" />
        <reference name="mytrips" />
        <reference name="cart"
              target="ShoppingCart/Updates" />
    </component>

    <component name="ShoppingCart">
        <implementation.java class=
              "com.tuscanyscatours.impl.ShoppingCartImpl"/>
        <service name="Updates" />
        <service name="Checkout" />
        <property name="currency" source="$currency" />
    </component>
</composite>
```

1 **Exposed services, references, and properties**

2 **Promoted services and references**

In listing 3.6 the `<composite>` element contains `<service>`, `<reference>`, and `<property>` elements 1 directly within the composite and outside the component

definitions that are also part of the composite. These elements define the composite services, composite references, and composite properties that are exposed by the Tours composite.

Each composite `<service>` and `<reference>` element has a `promote` attribute that identifies a component service or reference ❷ that will be exposed outside the composite when the composite is used as a component implementation. For example, the `<service>` element for BookTrip has a `promote` attribute with a value of `"TripBooking/Bookings"`, which means that the Bookings component service is visible externally as the BookTrip composite service. The Updates component service isn't promoted, which means it's visible only within the Tours composite.

There's no `promote` attribute for composite properties. Instead, the component property gets its value from a composite property by naming a composite property in the `source` attribute of the `<property>` element using an XPath expression. For example, the `currency` property of the ShoppingCart component has a `source` attribute of `$currency`, which means it gets its value from the `currency` composite property. This is different from service and reference promotion because the relationship between the component property and the composite property is defined within the component and not at the composite level. The composite property definition specifies a default value of `"USD"`, which means that the component property will have this value unless it's overridden in the definition of the higher-level component that uses the composite as its implementation.

Next we'll look at how to create a component that uses the Tours composite as its component implementation. In figure 3.13 the MyTours component has the Tours composite as its implementation. The component services, references, and properties of the MyTours component are the same as the composite services, references, and properties of the Tours composite that you saw in figure 3.12. The ToursImpl composite also contains another component TripProvider, and the `trips` reference of the MyTours component is wired to the Trips service of the TripProvider component.

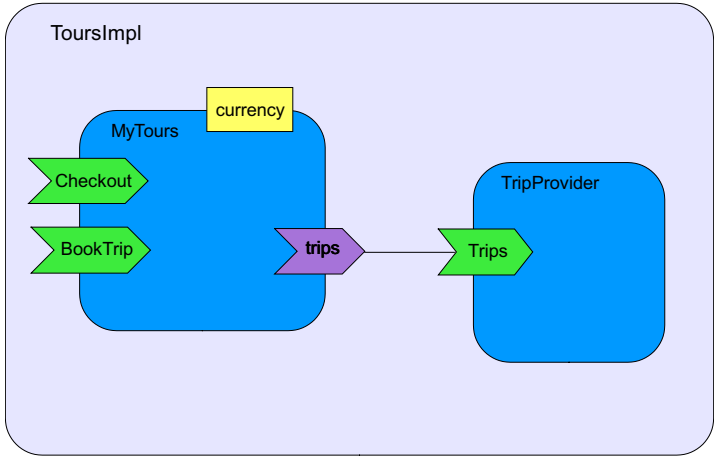


Figure 3.13 The Tours composite is used as the implementation of the MyTours component in the ToursImpl composite.

The next listing shows the XML definition for the ToursImpl composite pictured in figure 3.13. You can find this composite in the travel sample's contributions/building-blocks directory as the file tours-impl.composite.

Listing 3.7 Using a composite as a component implementation

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           xmlns:bb="http://bb.tuscanyscatours.com/"
           name="ToursImpl">

    <component name="MyTours">
        <implementation.composite name="bb:Tours" />
        <reference name="trips" target="TripProvider/Trips" />
        <property name="currency">GBP</property>
    </component>

    <component name="TripProvider">
        <implementation.java
              class="scatours.impl.TripProviderImpl" />
    </component>
</composite>
```

The implementation type of the MyTours component is `implementation.composite` with the XML qualified name of the Tours composite from listing 3.6. This means that the composite services, references, and properties of the Tours composite are used as the component services, references, and properties of the MyTours component.

As with other implementation types, the component definition can customize the services, references, and properties of its implementation. In this example the MyTours component has its `trips` reference wired to the Trips service of the TripProvider component. The `currency` property of the MyTours component is set to the value `"GBP"`, which overrides the default value of `"USD"` defined in the Tours composite. The TripProvider component used here is a scaffolded test version, which doesn't contain the production implementation of the Trips service.

You've seen what you can do with the black-box style of reuse, where composites are used as component implementations. Next we'll look at the white-box approach of including composites in other composites.

3.4.3 Including composites in other composites

Another way to reuse composites is by including them in other composites. This is equivalent to the include facilities in many programming languages, which bring in the contents of a separate file and treat the contents as if they had been placed inline.

We'll use the previous example to illustrate the value of composite inclusion. When we've finished testing the MyTours component, we'd like to connect it to the production version of the TripProvider component. This component is part of an existing composite named Trips in the trips.composite file, which you first saw in listing 1.3. We could copy the production TripProvider component definition from the Trips composite into the ToursImpl composite, but there's a better way to handle this.

By using composite inclusion, we can reuse the existing Tours composite directly, as shown in the following ToursImplInclude composite. You can find this composite in the travel sample's contributions/buildingblocks directory as the file tours-impl-include.composite.

Listing 3.8 Including a composite in another composite

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           xmlns:bb="http://bb.tuscanyscatours.com/"
           xmlns:gvt="http://goodvaluetrips.com/"
           name="ToursImplInclude">

    <component name="MyTours">
        <implementation.composite name="bb:Tours" />
        <reference name="trips" target="TripProvider/Trips" />
        <property name="currency">GBP</property>
    </component>

    <include name="gvt:Trips" />
</composite>
```

1 Trips included by ToursImplInclude

In listing 3.8 the ToursImplInclude composite contains an `<include>` element 1 for the Trips composite. This `<include>` element is replaced by the entire contents of the Trips composite, except for its outer `<composite>` element, as shown in the following listing.

Listing 3.9 Result of including a composite

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           xmlns:bb="http://bb.tuscanyscatours.com/"
           xmlns:gvt="http://goodvaluetrips.com/"
           name="ToursImplInclude">

    <component name="MyTours">
        <implementation.composite name="bb:Tours" />
        <reference name="trips" target="TripProvider/Trips" />
        <property name="currency">GBP</property>
    </component>

    <component name="TripProvider">
        <implementation.java class=
              "com.goodvaluetrips.impl.TripProviderImpl" />
        <service name="Trips">
            <interface.java interface=
                  "com.goodvaluetrips.Trips" />
        </service>
    </component>
</composite>
```

TripProvider added to ToursImplInclude

Listing 3.9 shows the result of the `<include>` directive in listing 3.8. The component definition for TripProvider from the Trips composite has been added to the ToursImplInclude composite in place of the `<include>` directive.

When using composite inclusion, the included composite doesn't need to have a complete external contract. For example, a component service defined in one included composite can be used as the target of a component reference defined in another included composite. It's also possible to include a composite that contains only wires, where the sources and targets of these wires are defined in other included composites.

3.4.4 Composite reuse in action

Composite implementations can be reused in any number of components, just like other implementation types such as `implementation.java`. For example, we can create components WSTours and JMSTours that expose the TripBook and Checkout services using Web Services and JMS bindings, as shown in the following listing. You can find this composite in the travel sample's contributions/buildingblocks directory as the file tours-appl.composite.

Listing 3.10 Reusing a composite implementation in different components

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           xmlns:bb="http://bb.tuscanyscatours.com/"
           xmlns:gvt="http://goodvaluetrips.com/"
           name="ToursAppl">

    <component name="WSTours">
        <implementation.composite name="bb:Tours" />
        <service name="BookTrip">
            <binding.ws uri=
                  "http://localhost:8081/BookTrip" />
        </service>
        <service name="Checkout">
            <binding.ws uri=
                  "http://localhost:8081/Checkout" />
        </service>
        <reference name="trips"
              target="TripProvider/Trips" />
    </component>

    <component name="JMSTours">
        <implementation.composite name="bb:Tours" />
        <service name="BookTrip">
            <binding.jms uri="jms:BookTrip" />
        </service>
        <service name="Checkout">
            <binding.jms uri="jms:Checkout" />
        </service>
        <reference name="trips"
              target="TourProvider/Tours" />
    </component>

    <include name="gvt:Trips" />
```

1 Same component implementation

2 Web service endpoints

3 Wired to Trips service

4 JMS endpoints

5 Wired to Tours service

```
    <component name="TourProvider">
        <implementation.java class=
              "com.budgettours.impl.TourProviderImpl" />
        <service name="Tours" />
    </component>
</composite>
```

There are a few interesting things going on in listing 3.10. The WSTours and JMSTours components both use the Tours composite from the tours.composite file as their implementations (1), with different customizations of the composite's exposed services and references. WSTours uses `binding.ws` to expose the services as web service endpoints (2), and JMSTours uses `binding.jms` to expose them as JMS endpoints (4). There's also a difference in how the components wire the `trips` reference. In WSTours this reference is wired to the Trips service (3) of the TripProvider component (implemented by the GoodValueTrips service provider and reused here by inclusion). In JMSTours the same reference is wired to a different service, Tours (5), in the TourProvider component from the BudgetTours service provider. The most interesting thing of all is that we've built this composite application by writing a single XML file without any new or changed implementation code.

Life, the universe, and composite applications

Now that we've shown how you can build a fully-fledged composite application using composites as building blocks, let's look at the main features that make it a composite application and not just an application.

First, the application is modular and componentized. The benefits of this are well known, so we don't need to dwell on them here. The modular parts (building blocks) of the ToursAppl application are the ToursAppl composite, the Tours composite, and the Java implementation classes `TripBookingImpl`, `ShoppingCartImpl`, `TripProviderImpl`, and `TourProviderImpl`. Each of these provides a contract through its services, references, and properties, and its internal implementation can be changed without affecting its external contract.

Second, the application uses different levels of composition. At the top level, the ToursAppl composite is composed from the Tours composite and the `TripProviderImpl` and `TourProviderImpl` implementations. At the next level, the Tours composite is composed from the `TripBookingImpl` and `ShoppingCartImpl` implementations. This ability to use one level of composition to build the next-higher level and continue doing this at multiple levels is called recursive composition, and it's a very important feature of SCA.

Third, each level of composition can include customization of the level below it. Without this, building a higher-level composition would often require us to make small changes to the building blocks that we're using. We've all seen the problems caused by this cut-and-paste style of reuse! SCA's ability to make customizations at the point of reuse means that existing building blocks can be used directly without needing to tweak them.

In this example you're starting to see the power and flexibility of using SCA to build composite applications. Using a few small XML files, we've taken some basic building blocks (Java implementation classes) and combined them in a particular way to make an application. By editing these XML files, we can update this application without needing to change any implementation code. Our application is modular and componentized, so we can change one part of it without affecting other parts.

In this section we've shown how composites can be reused within composite applications through inclusion in other composites or by using a composite to provide the implementation of a component. The inclusion approach allows for white-box reuse, where the internals of the composite being included are exposed and expected to be understood. The use of the composite as a component implementation supports a black-box approach, where the contents of the composite are hidden and not expected to be understood.

3.5 Summary

If you feel like a traveler who's just done Europe in a week, we're not surprised! We've covered a lot of ground in these first three chapters. In chapters 1 and 2 we introduced Tuscany and looked at the basic concepts of SCA, explaining and illustrating how you can use components, implementations, services, interfaces, references, wires, properties, and bindings to create a composite application. In this chapter, we completed the picture by looking at the detailed steps of packaging and running a composite application using Tuscany, and we showed how composite applications can be constructed from reusable building blocks.

The most important lesson to remember from this chapter is that to run an application in SCA and Tuscany, you need to describe your application's components and services as a composite application using a composite file. We also showed you how these composite files and related artifacts such as Java classes, XML documents, XSD files, and WSDL files are packaged into SCA contributions. Contributions are installed into the SCA domain, which is configured either as a single process or using multiple processes with the Tuscany domain manager. Composites can be deployed into the SCA domain, used as component implementations, or included in other composites.

Now it's time for a change of pace as we explore what Tuscany has to offer in greater depth and take a closer look at some of the important parts of Tuscany. We'll start by examining the different interaction patterns you can use in your services. Enjoy!

Part 2

Using Tuscany

Part 1 gave you a high-level view of most of the features of Apache Tuscany's SCA Java runtime. With the foundations laid, part 2 gives more detail about creating SCA components to support the various parts of your enterprise application.

Part 2 starts with chapter 4, "Service interaction patterns," which provides an overview of the different interaction patterns used to send messages from one SCA component to another and back again. You'll learn how the interacting components can run within the same JVM or in separate JVMs and how different styles of message exchange are configured. With SCA you can, depending on your requirements, choose whether components exchange messages synchronously or asynchronously, whether one message gives rise to a callback message, and whether a sequence of messages is grouped together in a conversation.

In chapters 5 and 6, "Implementing components using the Java language," and "Implementing components using other technologies," you'll be introduced to some of the technologies that can be used to implement components when using Apache Tuscany. Chapter 5 concentrates on Java and demonstrates how SCA components can be implemented using your existing Java skills. The lightweight SCA Java annotations and API are described, giving you access to a more comprehensive set of SCA features and interaction patterns. Chapter 6 explains how the SCA programming model is presented in BPEL, Spring, and Java-based scripting languages.

In order for components to interact with other components or with non-SCA applications, communication protocols must be defined in the form of SCA bindings. Chapter 7, "Connecting components using bindings," describes a number

of different bindings including Web Services, CORBA, RMI, JMS, and EJB, covering common communication protocols.

The frontend user interface will be an integral part of your service-based application. In chapter 8, "Web clients and Web 2.0," you'll learn about the implementation and binding types that Tuscany supports for integrating web-based applications with SCA services. This includes servlets, JSPs, Ajax, Static HTML, Atom, and RSS.

The data that flows between components can be presented in various formats. The Apache Tuscany runtime has a flexible and extensible framework for transforming data from one format to another. Chapter 9, "Data representation and transformation," describes this framework, giving examples of Java component implementations that use JAXB and SDO.

Chapter 10, "Defining and applying policy," describes how the SCA policy framework allows you to control quality of service features by applying policy to your composite application. You'll learn how, by using policy, you can control infrastructure features without requiring specific logic in your component implementations.

4 Service interaction patterns

This chapter covers

- Remote interaction
- Local interaction
- Request response interaction
- One-way interaction
- Conversational interaction
- Callback interaction

Components may interact with one another in different ways. It's similar to the way we interact with other people. For example, when Mary books a holiday using a travel agent, she may decide to visit them and talk to them directly, she may have to wait for them to call her back with some information, and she will certainly have an ongoing conversation about the holiday she's booking.

SCA defines common service interaction patterns and makes it simple to create component services and references that exploit them. In this chapter we'll start by introducing the different interaction patterns that SCA and Tuscany support. Then

we'll look at each pattern in more detail and explore the motivation, configuration, and applicability of each pattern.

We'll demonstrate each SCA interaction pattern by using a component from the travel-booking application. We're going to use the Hotel, Calendar, CurrencyConverter, and ShoppingCart components here. These components can be found in the following sample contributions, respectively:

- `contribution/hotel`
- `contributions/calendar`
- `contributions/currency`
- `contributions/shoppingcart`

The components don't work in isolation, so for each interaction pattern we've written a simple client component. For example, the InteractionLocalClient component demonstrates local interactions by sending a local message to the Calendar component. All the samples for this chapter are run with a single launcher from the launchers/interaction directory. The name of the client and target component for the interaction pattern being discussed is given so you can find the sample code easily. To run the example you'll need the following contributions as well as the contributions listed previously:

- `contributions/common`
- `contributions/interaction-service-remote`
- `contributions/interaction-client`

In this chapter we'll concentrate on the generic patterns, although we'll use the Java language in the samples to demonstrate these patterns. The details of how to configure and control these interaction patterns for Java implementations and interfaces are given in the next chapter (chapter 5). In Tuscany currently, support for the more complex interaction patterns is only provided by the implementation.java type.

By the end of this chapter, you'll have good understanding of the service interaction patterns described by SCA and will know how to use Tuscany to enable these patterns. This knowledge will help you choose suitable patterns for your application. Let's get started by first looking at the range of SCA interaction patterns.

4.1 Understanding the range of SCA service interaction patterns

SCA and Tuscany have been designed to allow you to exploit a wide variety of service interaction patterns between SCA components. The first two patterns describe the locality of the interacting component.

SCA components can have local or remote service interfaces. Services with local interfaces are able to communicate able only with other components that are deployed locally and that are running within a single JVM. On the other hand, services with remote interfaces can be deployed remotely and are able to communicate with other components, either in the same JVM or in different JVMs. Local and remote service interfaces exhibit different semantics. Local interfaces are pass-by-reference while remote interfaces are pass-by-value.

- *Remote*—The interacting components are designed to run in two different Tuscany nodes, in the same or separate Java virtual machines (JVMs), and could be running on different physical computers. Data passed between remote components is passed by value.
- *Local*—The interacting components are designed to run only in the same node and JVM. The Tuscany runtime uses in-memory communication in this case. Data passed between local components is passed by reference.

Regardless of whether components are local or remote, the following set of interaction patterns describes the style of each message exchange:

- *Request response*—The calling component sends a message to the called component. This is referred to as the *forward direction*. The calling component expects an immediate response for each message that's sent.
- *One way*—The calling component sends a message and doesn't wait for a response.
- *Conversational*—The calling component sends a sequence of related messages that are associated with each other by means of a common context maintained between calling and called components. When no conversation is in effect, each message is stateless and isn't related to the messages that come before or after it.
- *Callback*—A forward message from calling to called component instigates a call from the called component back to the calling component at some point in the future. This is referred to as the *callback direction*. When no callback is configured, messages travel in the forward direction only.

Some simple examples will help explain these patterns. First, imagine a simple SCA component service on the internet that provides a description of a holiday resort on request. The service has a single operation that accepts the request, looks up the resort details in a database, and returns them directly.

- Is the target service local? No, it's accessible across the internet (remote).
- Will the service return a direct response? Yes, the resort details are returned directly (request response).
- Is the request related to other requests? No, each request for information is separate (stateless).
- Will the service call back? No, no further action is expected once details are returned (forward only).

Now consider a more complex SCA component service that's able to find a resort that has available hotel rooms. This operation takes a little time, and so the service is designed to call back to the calling component once the information is available. The single operation in this service interface doesn't provide a result directly.

- Is the target service local? No, it's accessible across the internet (remote).
- Will the service return a direct response? No (one way).

- Is the request related to other requests? No, each request for information is separate (stateless).
- Will the service call back? Yes, once availability has been established, the calling component will be notified (callback).

Using this question-and-answer style is a useful approach to identifying when each pattern is applicable. Table 4.1 shows a set of questions that help demonstrate how the various patterns that SCA supports can be applied. The section numbers are included so that you can easily find the pattern in question. Because you don't need to use all of these patterns all of the time, you don't necessarily need to read all of the details in this chapter the first time through. You can use this table to help you pick and choose which sections are of interest.

Table 4.1 Questions that help you identify which service interaction patterns are appropriate and the sections where the patterns are documented

Question	Answer	Pattern	Section
Is the called component running in a different SCA runtime (Tuscany node) when compared to the calling component making the request?	Yes	Remote	4.2
	No	Local	4.3
Do you expect to get a direct response to the request?	Yes	Request response	4.4
	No	One way	4.5
Is the reference component expecting to be called back in the future as a result of the request?	Yes	Callback	4.6
	No	Forward only	4.4
Is the request related to a previous request to the same service?	Yes	Conversational	4.7
	No	Stateless	4.4

You can combine these patterns. For example, you could use one-way messages on the forward leg of a callback.

In the following sections we'll look at the details of each pattern. We'll first look at the remote and local interaction patterns, given that the other patterns apply equally to either of these.

4.2 Remote interaction

In this section we'll show how the remote interaction pattern allows SCA components to communicate with other SCA components or non-SCA software using pass-by-value semantics. The remote interaction pattern is vital to assembling coarse-grained, loosely coupled, and distributed components.

Coarse-grained components are typically responsible for carrying out large pieces of processing such as the TuscanySCATours Hotel component that interacts with business partners to find available hotel rooms. Such coarse granularity leads to loosely coupled components where message exchanges tend to comprise self-contained and

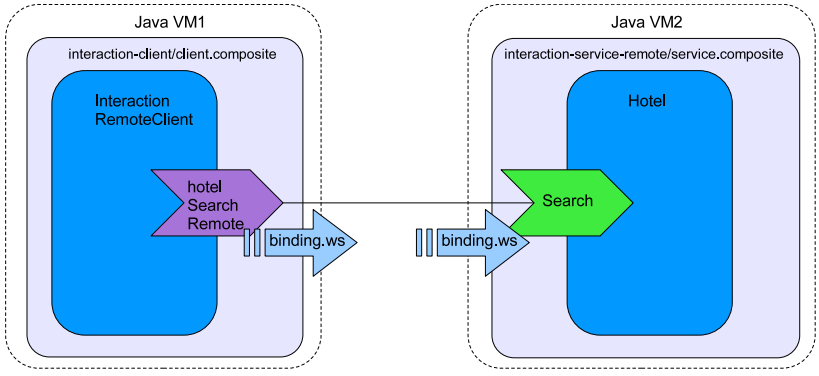


Figure 4.1 Remote interactions between components in separate Java virtual machines. The components communicate using the web service binding.

independent messages. This gives the components great flexibility in how, where, and when to complete the tasks they've been asked to perform.

Figure 4.1 shows a client component wired to the Hotel component in order to search for available hotel rooms. The two components are running in separate JVMs, which can be running on separate computers in different geographical locations.

Coarse-grained components usually run remotely from the calling component. Often this is because they're developed and operated independently by different parts of an organization or even different organizations. In the TuscanySCATours travel application, the Hotel component is implemented by a department that specializes in working with hoteliers. It's made available over the web services binding for any part of the organization to call.

Configuring a component in SCA to allow its service to be accessed remotely is a simple matter of marking the service interface appropriately.

4.2.1 Configuring remote interaction

For service interfaces described using Java interfaces, SCA defines a Java annotation `@Remotable`. This annotation indicates that component services that implement the Java interface can be remote and that any data passed to the service will be passed by value. The following code, which can be found in `contributions/common`, shows how the `Search` interface is configured to be remotable:

```
@Remotable
public interface Search {
    TripItem[] searchSynch(TripLeg tripLeg);
}
```

The Hotel component, which implements this `Search` interface, can run anywhere on the network. Bindings such as web service or JMS bindings can then be configured to allow clients to communicate with the Hotel component.

SCA bindings implement protocols for exchanging information between component services across a network. For example, the web service binding uses SOAP over

HTTP to transfer messages in XML-formatted SOAP envelopes. Tuscany provides several bindings, as described in chapter 7.

The following code shows the InteractionRemoteClient component reference `hotelSearchRemote` configured with a web service binding pointing to the Hotel component Search service:

```
<component name="InteractionRemoteClient">
    <implementation.java class=
        "scatours.client.impl.InteractionRemoteClient" />
    <reference name="hotelSearchRemote">
        <binding.ws uri="http://localhost:8081/Hotel/Search"/>
    </reference>
 </component>
```

Note that the reference name `hotelSearchRemote` is just a name. The inclusion of the word `Remote` here is to remind us that this sample is about remote interaction. This is true for the other samples in this chapter where the name of the interaction pattern is included in the reference name.

A separate composite describes the Hotel component as follows:

```
 <component name="Hotel">
    <implementation.java class="com.tuscanyscatours.hotel.impl.HotelImpl"/>
    <service name="Search">
        <binding.ws uri="http://localhost:8081/Hotel/Search"/>
    </service>
 </component>
```

If you choose to use `interface.wsdl` to describe a service interface in the composite file, as discussed previously in section 2.2.2, then the service you describe is assumed remotable.

4.2.2 Exploiting remote interaction

The developer of a client component doesn't need to worry about whether the Hotel component is running in the same JVM or not. The Tuscany infrastructure handles the details of marshalling the message to the network and unmarshalling the response.

The term *marshalling* used here means the conversion of a message in the form in which it appears in memory into the form in which it appears in a communication protocol, for example, converting from a JavaBean to XML. Unmarshalling means the opposite, where the message is converted back from its communications protocol form to the form it takes in memory.

Here's the implementation of the InteractionRemoteClient component. The `getTestTripLeg()` method is omitted because it just creates a `TripLeg` structure.

```
public class InteractionRemoteClient implements Runnable, SearchCallback{

    @Reference
    protected Search hotelSearchRemote;

    public void run() {
       TripLeg tripLeg = getTestTripLeg();
```

```
        TripItem[] tripItems = hotelSearchRemote.searchSynch(tripLeg);
    }
}
```

The InteractionRemoteClient component talks to the Hotel component using the `hotelSearchRemote` reference proxy. This proxy is injected into the InteractionRemoteClient implementation by the Tuscany runtime based on the definition of the component in the composite file. In this case, this means that the proxy will use SOAP/HTTP web services to call the Hotel component using the address http://localhost:8081/Hotel/Search.

The implementation of the Hotel component has no notion that the client that's sending the request is remote. At the service end, the Tuscany infrastructure handles the details of unmarshalling the message from the network and marshalling the response.

```
public class HotelImpl implements Search{
    public TripItem[] searchSynch(TripLeg tripLeg) {
        List<TripItem> items = new ArrayList<TripItem>();

        for(HotelInfo hotel : hotels){
            if (hotel.getLocation().equals(tripLeg.getToLocation())){
                TripItem item = new TripItem(hotel);
                items.add(item);
            }
        }

        return items.toArray(new TripItem[items.size()]);
    }
}
```

The implementation is just a normal Java class implementing the `Search` Java interface. No additional work is required of the implementation developer.

As we described in chapter 2, messages passed to remotable services are passed by value. Pass-by-value semantics are enforced even if the remotable component service happens to be running in the same process as the calling component. Hence the behavior of the composite is predictable regardless of where the components run.

If a Java service interface is not marked with `@Remotable`, then the service is configured by the Tuscany runtime to take part in only local interactions.

4.3 Local interaction

The local interaction pattern is useful for connecting components that are located within the same JVM. Unnecessary network traffic can be avoided because communication passes directly from one component to the next without the need to marshal messages to and from the network.

Figure 4.2 shows a client component wired to a Calendar component, which in turn provides operations that manipulate date values. In this particular example, the InteractionLocalClient component and Calendar component are wired within a single JVM and are local to one another.

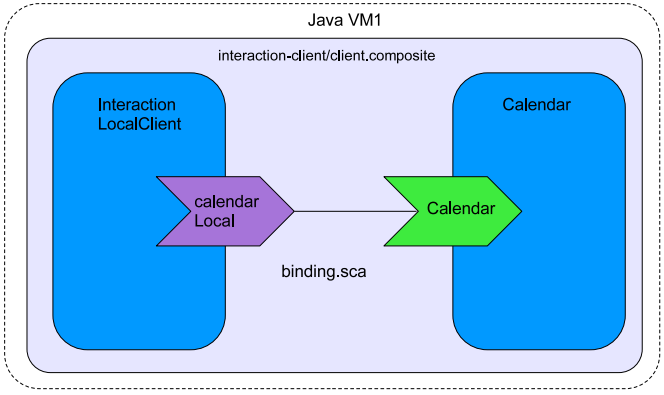


Figure 4.2 Local interactions between components within a single Java virtual machine

The local interaction pattern is often used to connect fine-grained components to create a coarse-grained component that in turn exposes a remote interface. For example, you might combine the Calendar component with a TripBooking component to provide a service that can answer date-oriented queries such as "find all the flights that leave on Monday." This kind of fine-grained composition can, of course, be done outside SCA. But you may choose to use SCA to present a consistent programming model regardless of whether you're using remote or local interaction.

4.3.1 Configuring local interaction

Local interaction is the default mode of operation in SCA when using Java component implementations, and so local Java service interfaces require no SCA-specific configuration.

In the following example, the local Calendar service interface is defined by a simple Java interface:

```
public interface Calendar {
    String getEndDate(String startDate, int duration);
}
```

Each component service defined by a Java interface without the SCA `@Remotable` annotation is by default assumed to use the local interaction pattern. If you choose to use WSDL interfaces to describe your component's service interface, then they can't be used to define local component interactions.

4.3.2 Exploiting local interaction

Tuscany supports the SCA concept of a default SCA binding, `binding.sca`, which will work for both local and remote services. The default binding is assumed to be active for any wires where no binding is explicitly specified. This makes wiring two local components straightforward. The following shows how the InteractionLocalClient component is wired to the Calendar component:

```
<component name="InteractionLocalClient">
    <implementation.java class=
```

```
        "scatours.client.impl.InteractionLocalClient" />
    <reference name="calendarLocal" target="Calendar"/>
</component>

<component name="Calendar">
    <implementation.java
         class="com.tuscanyscatours.calendar.impl.CalendarImpl"/>
</component>
```

Note that no bindings are specified on the reference of the InteractionLocalClient component. Also note that no service is explicitly specified here for the Calendar component. The Tuscany runtime uses `binding.sca` configuration when no binding is provided, and this is enough to connect the two components.

The following code shows a cut-down version of the InteractionLocalClient component's implementation. The `getTestTripLeg()` method is omitted because it just creates a `TripLeg` structure:

```
public class InteractionLocalClient implements Runnable {

    @Reference
    protected Calendar calendarLocal;

    public void run() {
       TripLeg tripLeg = getTestTripLeg();
       String toDate = calendarLocal.getEndDate(tripLeg.getFromDate(), 10);
       tripLeg.setToDate(toDate);
    }
}
```

Looking at the implementation of the client component, you see the calendarLocal field. A proxy to the Calendar component Calendar service is injected into this field by the Tuscany runtime. The `run()` method uses the calendarLocal proxy to communicate with the Calendar component.

Remember from chapter 2 that if your service interface is local, it can be wired only to references of components running in the same Tuscany node using `binding.sca`. This may be useful for fine-grained composition where pass-by-reference is the norm. But you can't then change your configuration and attach bindings, such as `binding.ws`, to the service and pass messages to it from components running in other Tuscany nodes. For this your service interface must be remotable.

Within an SCA domain, the Tuscany implementation of `binding.sca` will ensure that messages reach a remotable service regardless of whether it's running in the same node or running in a remote node. This brings a lot of flexibility to your application. In this way, components can be deployed in different configurations within the SCA domain without having to change the binding configuration within the composite.

Regardless of whether a component service is remote or local to the component that's trying to call it, the two interacting components need to exchange information. Let's now move on to look at the first interaction pattern that builds on top of local and remote communication: request response.

4.4 Request response interaction

Once two services are wired together, either locally or remotely, they're ready to begin to communicate with one another. When a component uses a reference to call another component's service, the next step in processing often depends on the response from the other component. In this situation, the requesting component is willing to, and must, wait for the response to come back before it proceeds.

For example, figure 4.3 shows a test client calling the CurrencyConverter component using the request response interaction pattern. The InteractionRequestResponseClient component calls the CurrencyConverter component and waits for the result to be retuned before continuing with its processing.

The normal SCA diagram style has been extended here with a dashed line that shows the flow of execution through the pair of components. The InteractionRequestResponseClient component implementation makes a request though the currencyConverterRequestResponse reference to the CurrencyConverter service. It then waits until the CurrencyConverterImpl component implementation has finished processing the request and has returned a response before continuing.

4.4.1 Configuring request response interaction

Request response is the default interaction pattern in SCA. No extra annotation is required to enable this mode of operation. Looking again at the interface for the CurrencyConverter component, you can see that this is just a normal Java interface:

```
public interface CurrencyConverter {

    double getExchangeRate(String fromCurrencyCode,
                           String toCurrencyCode);
}
```

The `getExchangeRate` method is expecting to take two `String` parameters, `fromCurrencyCode` and `toCurrencyCode`, do its processing, and return the resulting exchange rate as a `double` value. A caller of this method will call `getExchangeRate` with the appropriate inputs and will then wait until the exchange rate is returned before continuing.

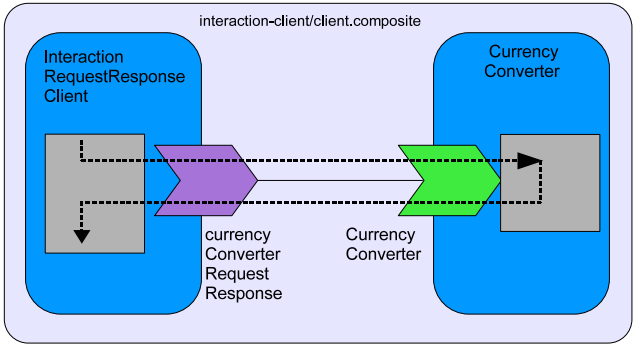


Figure 4.3 A request response message flowing between the InteractionRequestResponseClient and CurrencyConverter components. The flow of execution is depicted using the dashed line. Program execution at the client waits until the request completes and the response is returned.

4.4.2 Exploiting request response interaction

The request response interaction pattern is familiar to most programmers. When writing a program in, say, the Java language or C++, a call to a method or function usually follows the request response style. The calling method pushes the call details onto the stack, the called method performs its intended function, and it's not until the called method completes that the stack pops and the calling method continues processing.

The pattern in SCA is much the same; no extra configuration is required to use it either in the component implementations or in the composite files. The CurrencyConverter component's service implementation is plain Java code, as follows:

```
public class CurrencyConverterImpl implements CurrencyConverter {

    public double getExchangeRate(String fromCurrencyCode,
                                  String toCurrencyCode){
        return rates[currencyIndex.get(fromCurrencyCode).intValue()]
                    [currencyIndex.get(toCurrencyCode).intValue()];
    }
}
```

The InteractionRequestResponseClientImpl that calls this service is also simple and requires only the SCA `@Reference` configuration to identify the reference proxy:

```
@Service(Runnable.class)
public class InteractionRequestResponseClient implements Runnable {

    @Reference
    protected CurrencyConverter currencyConverterRequestResponse;

    public void run() {
      double exchangeRate =
            currencyConverterRequestResponse.getExchangeRate("GBP", "USD");
    }
}
```

The request response interaction pattern is not appropriate in many cases. For example, imagine you want to extend the TuscanySCATours application to allow a request to be sent for a set of paper brochures to be delivered to the customer. This doesn't lead immediately to the brochures' being delivered electronically by return. The brochure-ordering service must arrange for appropriate brochures to be collated and delivered to the customer's address. An immediate response is not required by the calling component, and a one-way pattern could be more appropriate.

4.5 One-way interaction

We've looked at the default request response interaction pattern where the caller waits for a response to a request. When the focus is on coarse-grained and loosely coupled services, the request response pattern doesn't necessarily fit well. A request response interaction can grow into a network of interactions if the called services also make request response interaction–style calls to other services. This extended graph

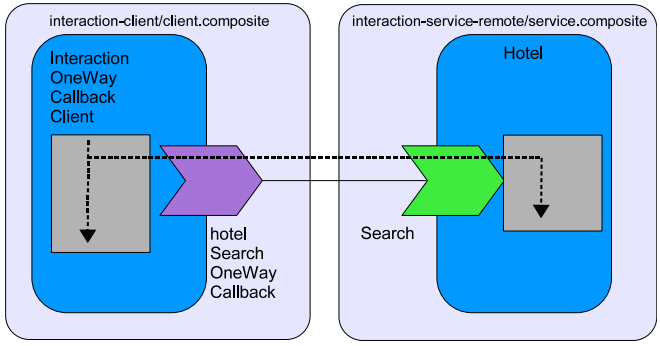


Figure 4.4 The InteractionOneWayCallbackClient component sends a search request to the Hotel component using a one-way interaction pattern. The flow of execution is depicted using the dashed line. This diagram doesn't show how the results are retrieved from the Hotel component.

of service calls, waiting for other services to complete their processing, is often at odds with the objective of using loosely coupled services.

SCA defines the one-way interaction pattern as an alternative to request response. Anyone who has used threads in the Java language or C++ applications will recognize that it's more attractive to arrange for processing to proceed in parallel. In this way, the calling processing can continue at the same time as the called processing.

Figure 4.4 shows a test-client component sending a request to the TuscanySCA-Tours Hotel component to perform a search.

The search processing may take a while to complete. Instead of waiting, the client component can move onto other things, such as searching for flights and cars. We'll talk about how to retrieve the response from the Hotel component when we cover the callback interaction pattern in the next section.

With the one-way pattern, we lose the temporal coupling between components in the call graph. A delay in one part of the graph won't automatically be multiplied because other components in the graph wait for the delayed service to complete.

Closely related to the problem of temporal coupling is the need to successfully manage the ever-changing load in an active application. When request response interaction is in force, it's hard to delay or move the processing at a specific component in order to smooth out processing fluctuations. With services operating independently, the details of where and when processing happens vary and are easier to control.

The flexibility gained by using the one-way pattern leads to less-fragile applications. For example, let's assume that the request response pattern is used when making a request to a Hotel component to search for available hotels. If a power failure stops the computer running the Hotel component, then the client component that's waiting for it to complete its processing will fail. Extra code is needed to detect such failures and retry the request.

If, on the other hand, the Hotel component is asked to process the request using the one-way pattern, the client component can continue processing. It'll be notified when the hotel search results are available. If in the meantime the Hotel component stops and is restarted, it can be coded to continue processing the original request. The client component won't be affected. In such a case, the one-way interaction pattern is more appropriate.

4.5.1 Configuring one-way interaction

Individual methods on a Java service interface can be marked as one way using the `@OneWay` annotation. No extra configuration is required in a composite file to use this pattern. The following code shows the `Search` interface extended to include both request response (`searchSynch`) and one-way (`searchAsynch`) operations for initiating a search.

```
public interface Search {
    TripItem[] searchSynch(TripLeg tripLeg);

    @OneWay
    void searchAsynch(TripLeg tripLeg);
}
```

The `@OneWay` annotation can be applied only to operations that don't return data. For example, in the Java language, this means operations that have a `void` return type and have no declared exceptions.

4.5.2 Exploiting one-way interaction

Anyone familiar with using message-oriented middleware will be familiar with the one-way interaction pattern. The calling component uses an SCA reference proxy to send a message to another component and doesn't wait for a response.

The Hotel component implements the `searchAsynch` operation using the same logic as the `searchSynch` operation because both operations return the same results.

```
public class HotelImpl implements Search{

    public void searchAsynch(TripLeg tripLeg) {
        TripItem[] items = searchSynch(tripLeg);
        ...
    }
}
```

Note that we don't show here what happens with the results. The InteractionOneWayCallbackClientImpl component implementation is also simple:

```
public class InteractionOneWayCallbackClientImpl implements Runnable {

    @Reference
    protected Search hotelSearchOneWayCallback;

    public void run() {
      TripLeg tripLeg = getTestTripLeg();
      hotelSearchOneWayCallback.searchAsynch(tripLeg);
      ...
    }
}
```

With the one-way version of the `searchAsynch` operation, you can see that no return data is expected by the client. You may be wondering how the results of the search are returned. One option is to configure the component that implements the `Search`

interface so that it can call back to the client component when the search is complete. That's why the client here is called the InteractionOneWayCallbackClient. We'll use the same example in the next section to show how callbacks work.

4.6 Callback interaction

SCA defines a model for calling back from target to client components that's configured by using callback interfaces. The combination of a forward interface with its callback interface is referred to as *bidirectional interfaces* in the SCA specifications. The full details of the Java language–specific callback programming model are described in chapter 5, but because Tuscany currently supports callbacks in only the Java component implementation, we'll use some Java code here to aid our description of the general pattern.

Using the callback interaction pattern, a target component can send a response back to the client component asynchronously. The client doesn't wait for the result; it gets notified when it's available. For example, a one-way operation doesn't allow a service to return data to the client component directly. As soon as the request is sent, the client component continues processing. At some point in the future, the target service will process the message. If a result is to be provided back to the client component, then a callback style of message exchange is required.

Even in the case of a request response–style operation, processing may have been kicked off at the target component that requires a call to be made back to the client component at some point in the future. It's often the case that the immediate response is used to indicate that processing has been successfully started, whereas the real results are delivered later using a callback.

Consider again the Hotel component from the TuscanySCATours application that we used when talking about the one-way pattern. In the TuscanySCATours application the TravelCatalog component, see `contributions/travelcatalog`, uses the Hotel component, see `contributions/hotel`, to search for available hotel rooms for a customer's trip. It may take the Hotel component some time to complete its search, and preferably the TravelCatalog should be notified when the request is processed instead of waiting for a response. In the meantime, the TravelCatalog component starts searching for flights and cars. In figure 4.5 we've pulled the Hotel component out of the TuscanySCATours application, and we'll show an InteractionOneWayCallbackClient component that we'll use here to demonstrate the callback interaction pattern alongside the one-way interaction pattern. We've wired this client component directly to the Hotel component. It's important to note that there's only one wire between the two components. The callback is handled by the Tuscany runtime without the need for a separate callback wire.

A component service that implements the callback interaction pattern has two interfaces: the forward interface and the callback interface. The forward interface is implemented by the target component. The callback interface is implemented by the client component to which the future callback will be directed.

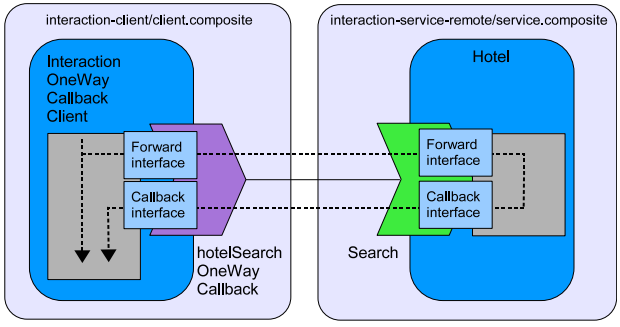


Figure 4.5 A bidirectional service combines both forward and callback interfaces. The flow of execution is depicted using the dashed line. The InteractionOneWayCallbackClient component makes a request through the forward interface to the Hotel component and continues processing immediately. The Hotel component processes the request and sends the results back via the callback interface.

Bidirectional service interfaces can be used for both local and remote interactions. In this case, the forward and callback interfaces must be either both local or both remotable.

4.6.1 Configuring callback interaction

To indicate that a component service has a bidirectional interface, the service interface must be annotated with the name of the interface through which the service will perform the callback. The following code shows part of the Java Search service interface, see `contributions/common`, that the Hotel component implements:

```
@Remotable
@Callback(SearchCallback.class)
public interface Search {
    @OneWay
    void searchAsynch(TripLeg tripLeg);
}
```

The `@Callback(SearchCallback.class)` annotation indicates that the services that implement this interface are bidirectional and will use the `SearchCallback` interface to perform their callbacks. The callback interface is a Java interface, as follows:

```
@Remotable
public interface SearchCallback {
    void searchResults(TripItem[] items);
}
```

There's nothing special about this interface, but the client component must implement it so that the component service can call back to it.

With this `@Callback` annotation, the Tuscany runtime has enough information to determine that a normal wire between a reference and a service describes a bidirectional interface without further configuration. The following component descriptions show how the client component and Hotel component could be wired to support a bidirectional interaction if the components were defined with default bindings:

```
<component name="InteractionOneWayCallbackClient">
    <implementation.java class=
        "scatours.client.impl.InteractionOneWayCallbackClient" />
```

```
    <reference name="hotelSearchOneWayCallback" target="Hotel"/>
    ...
</component>

<component name="Hotel">
  <implementation.java class="com.tuscanyscatours.hotel.impl.HotelImpl"/>
</component>
```

The `hotelSearchOneWayCallback` reference targets the Hotel component in exactly the same way as the forward-only case. When this composite file is loaded into the Tuscany runtime, the following steps are taken to configure the wire between the InteractionOneWayCallbackClient component and the Hotel component:

1. The hotel component service endpoint is created using the default SCA binding (`binding.sca`).
2. A callback service endpoint is added to the InteractionOneWayCallbackClient component and given the name `SearchCallback`. This too will use the default SCA binding and is the service that receives callback messages from the target service.

This composite configuration relies on the defaults that SCA assumes to be in force. When it comes time to deploy components in your environment, you may want to control the callback message path in more detail. SCA allows you to configure the interface for the callback service as well as the binding that will be used for callbacks.

Figure 4.6 shows the forward and callback interfaces and bindings and the role they play in bidirectional interaction.

Bidirectional interface configuration is achieved by adding a `callbackInterface` attribute to the reference and service interface elements. The interface can be described using any of the supported interface styles, currently Java interface and WSDL. The following code shows how a bidirectional interface can be specified in the composite file using the Java interface description style:

```
<interface.java interface="com.tuscanyscatours.common.Search"
            callbackInterface="com.tuscanyscatours.common.SearchCallback"/>
```

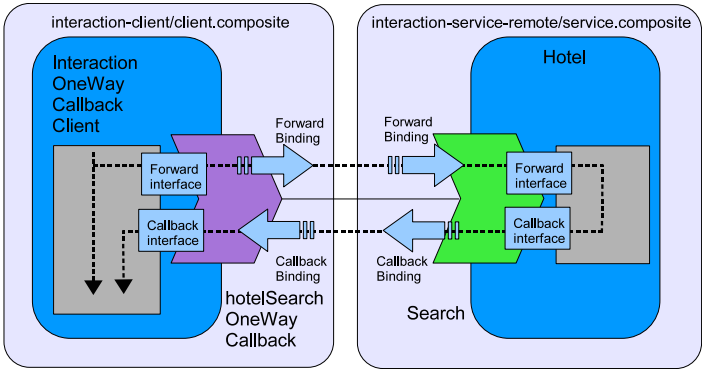


Figure 4.6 The forward and callback interfaces and bindings in bidirectional interfaces. The flow of execution is depicted by the dashed line.

Bidirectional binding configuration is achieved by adding a `<callback/>` element to reference and service elements in the composite file. The following code snippet shows how to control the forward and callback bindings for a service:

```
<service name="Search">
  <binding.ws />
  <callback>
    <binding.jms/>
  </callback>
</service>
```

The forward and callback bindings don't have to match. This can be useful if callback messages need to be separated from forward messages.

At the reference, the configuration takes a similar form. The following code shows how to configure the reference in order to control the forward and callback bindings. Again, the forward and callback bindings can be different:

```
<reference name="hotelSearchOneWayCallback">
  <binding.ws />
  <callback>
    <binding.jms/>
  </callback>
</reference>
```

In our example the Hotel component is remote and is running in a separate node to the InterfaceOneWayCallbackClient. The web services binding is used for both forward and callback bindings. The component definition for the Hotel component is as follows:

```
<component name="Hotel">
    <implementation.java class="scatours.hotel.HotelImpl"/>
    <service name="Search">
        <interface.java interface="com.tuscanyscatours.common.Search"
              callbackInterface="com.tuscanyscatours.common.SearchCallback"/>
        <binding.ws uri="http://localhost:8081/Hotel/Search"/>
        <callback>
            <binding.ws uri="http://localhost:8080/Client/SearchCallback"/>
        </callback>
    </service>
</component>
```

The `<interface.java callbackInterface=""/>` attribute explicitly describes what the callback interface is expected to be. In this case, the interface is `com.tuscanyscatours.common.SearchCallback`.

The `<callback/>` element contains the binding configuration associated with the callback for the Search service. Here both forward and callback messages are delivered using the web services binding. The callback binding description at the service effectively describes a reference to the callback service.

The reference that exploits the bidirectional interface on the Hotel component is configured as follows:

```
<component name="InteractionOneWayCallbackClient">
    <implementation.java class=
        "scatours.client.impl.InteractionOneWayCallbackClient" />
    <reference name="hotelSearchRemote">
        <interface.java interface="com.tuscanyscatours.common.Search"
            callbackInterface="com.tuscanyscatours.common.SearchCallback"/>
        <binding.ws uri="http://localhost:8081/Hotel/Search"/>
        <callback>
            <binding.ws uri="http://localhost:8080/Client/SearchCallback"/>
        </callback>
    </reference>
 </component>
```

Again, both the callback interface and binding are explicitly described. At the reference the callback binding effectively describes a service. At runtime this information is used to create a callback service on the reference component. In Tuscany the name of this service will be the name of the reference that defines it, in this case `hotelSearchRemote`. But there's no need to explicitly refer to the callback service because the runtime ensures that callbacks arrive at the right component.

4.6.2 Exploiting callback interaction

To send a callback message, the service implementation has to obtain a proxy that points back to the calling component. When using a Java implementation, you can retrieve the proxy to the callback service by adding the SCA `@Callback` annotation to the field into which you want Tuscany to inject the callback proxy.

```
public class HotelImpl implements Search {

    @Callback
    protected SearchCallback searchCallback;

    public void searchAsynch(TripLeg tripLeg) {
        TripItem[] items = searchSynch(tripLeg);
        searchCallback.searchResults(items);
    }
}
```

The InteractionOneWayCallbackClient that calls the Hotel component must implement the callback interface, in this case `SearchCallback`, so that the Hotel component can call back to it.

```
public class InteractionOneWayCallbackImpl implements Runnable,
                                                       SearchCallback{
  @Reference
  protected Search hotelSearchOneWayCallback;

  public void run() {
```

```
    TripLeg tripLeg = getTestTripLeg();
    TripItem[] tripItems = hotelSearchOneWayCallback.searchAsynch(tripLeg);
    ...
  }

  public void searchResults(TripItem[] items){
    ...
  }
}
```

In this example, following the call to `searchAsynch`, the Hotel component calls back to the `searchResults` operation on the IteractionOneWayCallbackClient component. It's then the client component's responsibility to process the results.

You now have a high-level understanding of how to build bidirectional interfaces and describe component wiring in order to configure callbacks for operations in your application.

At the moment, Tuscany SCA supports the callback mechanism only in the Java implementation type. We'll describe the details in chapter 5. You can, of course, simulate the callback interaction pattern when it's not directly supported by Tuscany. To do this you'll need a service on the target component to which the forward reference can be wired. The client component must also provide a service to which a callback reference from the target component can be explicitly wired. Generation and maintenance of an ID to tie the callback call to a forward call is the responsibility of the application.

Now let's move on and look at how to correlate a series of messages into a single conversation.

4.7 Conversational interaction

The conversational interaction pattern describes how operations in a service interface are associated with one another. As with callbacks, Tuscany supports the conversational interaction pattern only in Java component implementations. The details of the Java programming model for conversations, including its many related Java annotations, are discussed in chapter 5. Here we'll omit many of the detailed annotations and take a high-level look to complete our list of interaction patterns.

> **NOTE** The future of conversational interfaces in the SCA specifications is under discussion. Although there are similar concepts in some other middleware technologies, there's currently no accepted industry standard for conversational semantics. Because of the lack of industry standardization in this area, conversations aren't included in the SCA 1.1 specifications being defined by OASIS and will be reconsidered for a future version of SCA. They're supported in the Tuscany SCA 1.x codebase but not in Tuscany 2.x.

Using the conversational interaction pattern, a component can process a sequence of messages as part of the same conversation and rely on the Tuscany runtime to start

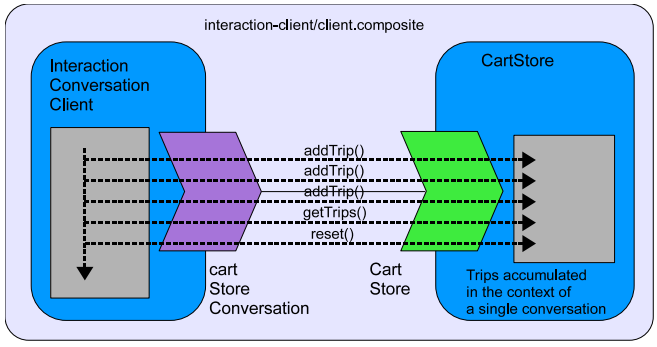


Figure 4.7 Multiple conversational calls to the CartStore component can be correlated so that each message is processed by the same component instance. The flow of execution is depicted using the dashed lines.

and end the conversation as defined by the service interface. For example, in the TuscanySCATours application, the ShoppingCart component, which can be found in `contributions/shoppingcart`, adds items to the CartStore component in order to build up the trip package as the customer selects trip items. The CartStore component, which can also be found in `contributions/shoppingcart`, manages these trip items as part of a single conversation.

To demonstrate conversations we've pulled the CartStore component out of the TuscanySCATours application and call it with the InteractionConversationClient. Figure 4.7 shows how the InteractionConversationClient sends multiple messages to the CartStore component in the context of a single conversation.

The CartStore component stores the contents of the customer's shopping cart for the trips the customer is trying to book. A conversation with the CartStore component begins when the first item is added.

As the client adds and removes trips, the Tuscany runtime ensures that messages sent to the customer's CartStore are all processed in the context of the same conversation. The messages belonging to a conversation may be seconds or days apart. Therefore, the conversation context should be identifiable over the period of time that the conversation is in effect.

The conversation continues, and the client can retrieve the list of trips currently in the store at any time. At some point the customer decides to purchase the items in the shopping cart, the CartStore is emptied, and the conversation ends.

Now let's look at how to configure conversational interactions in SCA applications.

4.7.1 Configuring conversational interaction

A Java service can take part in a conversation if the service interface is marked with the `@Conversational` Java annotation. The following listing shows the CartStore service interface.

Listing 4.1 Configuring conversational interaction

```
@Remotable
@Conversational
public interface CartStore {
```

```
    void addTrip(TripItem tripItem);

    void removeTrip(TripItem trip);

    TripItem[] getTrips();

    @EndsConversation
    void reset ();
}
```

Start or continue conversation

The `@Conversational` annotation is shown just above the interface definition. This means that the first call to any of the interface's operations will start a conversation. The conversation will continue as the client calls subsequent operations. The conversation ends when the operation annotated with `@EndsConversation` is called. No extra configuration is required in the composite files in order to use the conversational interaction pattern.

4.7.2 Exploiting conversational interaction

The Tuscany runtime tracks related requests that are part of the same conversation by associating a conversation ID with each request. The conversation ID is used to automatically locate the component instance associated with each ongoing conversation. Because each component instance is associated with only one conversation, the component implementation can use this knowledge to store and retrieve conversation state as it sees fit.

Using a Java language example, a component that's intended to participate in a conversation must include the `@Scope` annotation that indicates `CONVERSATION` scope. `CONVERSATION` scope ensures that each component instance is associated with a separate conversation ID. Because this annotation is Java language specific, it's described in more detail in chapter 5. Conversations won't work if you don't configure your service with `CONVERSATION` scope because multiple threads for different conversations could be passing through the same component instance, and hence the state of the component instance couldn't be tied to a single conversation.

The following code shows how the CartStore component implementation uses `CONVERSATION` scope.

Listing 4.2 The CartStore component implementation with `CONVERSATION` scope

```
@Scope("CONVERSATION")

@Service(interfaces={CartStore.class})
public class CartStoreImpl implements CartStore{

    private List<TripItem> trips = new ArrayList<TripItem>();

    public void addTrip(TripItem trip) {
        trips.add(trip);
    }
```

New instance for each conversation

```
    public void removeTrip(TripItem trip) {
        trips.remove(trip);
    }

    public TripItem[] getTrips(){
        return trips.toArray(new TripItem[trips.size()]);
    }

    public void reset(){
        trips.clear();
    }
}
```

When a message arrives at a `CONVERSATION`-scoped component, the Tuscany runtime automatically uses the conversation ID contained in the incoming message to locate the correct instance of the target component. It then passes the incoming message to this instance. This process is repeated as each message arrives for the conversational component until the conversation ends and the component instance is removed.

The writer of the component implementation needn't be concerned with how the conversation is identified. The conversation ID is available in the Java component implementation if required, using the `@ConversationID` annotation. See chapter 5 for details.

The Tuscany runtime doesn't persist any conversational state or conversation IDs automatically. If the runtime is stopped or crashes, then conversational state will be lost. It's the responsibility of the application developer to persist conversational state as it accumulates, if required.

In a component implemented using the Java language, you can use your favorite Java persistence technology to store a conversational component instance's state data. For example, you could use JDBC to store all component member data each time it changes. In that way, if the runtime is stopped, you can arrange for the state to be read back from the database when the application is restarted.

You now know how to treat a sequence of messages as a single conversation. It's important to point out that neither the conversational nor the callback interaction patterns preclude you from passing conversational and callback-style information along with application data in order to implement these patterns manually if you prefer. The Tuscany runtime supports these patterns but doesn't mandate their use.

4.8 Summary

Tuscany and SCA are all about allowing you to describe services and wire them together into composite applications. In this chapter you've seen that SCA and Tuscany provide a great deal of flexibility in how such applications can be constructed.

You can control the location of components involved in the composite application. They can be local, in the same process, or remote. SCA components can interact with one another either synchronously or asynchronously, depending on your business requirements.

When the asynchronous pattern is appropriate, you have the choice of using one-way interfaces and callbacks to manage the way that messages are exchanged.

Finally, conversations allow a component to relate one incoming message to another so that state can accumulate as a business process runs and messages are exchanged between components.

The interaction patterns described in this chapter provide a common base for all SCA composite applications, and you'll notice that they're used in various examples throughout the book. The Java implementation type, implementation.java, has the best-developed support for the various interaction patterns we've discussed here. We'll cover the details of how to control each interaction pattern in components implemented using the Java language in the next chapter.

5 Implementing components using the Java language

This chapter covers

- SCA Java annotations
- Java interfaces for services and references
- Services, references, and properties in Java implementations
- Component scope
- Callbacks and conversations
- Passing SCA service references
- Error handling

Each component in a Tuscany SCA composite application is implemented using an implementation type. The SCA specifications define a number of implementation types, for example:

- implementation.java
- implementation.spring
- implementation.bpel

The Tuscany project has also added a few more, for example, implementation.script.

In this chapter we'll focus on implementation.java. The other implementation types mentioned will be discussed in chapter 6.

The SCA Java Component Implementation specification defines implementation.java (http://www.osoa.org/download/attachments/35/SCA_JavaComponentImplementation_V100.pdf). This implementation type allows application developers to use new or existing Java classes and interfaces to implement SCA components. These components can then be wired with other components, either locally or remotely, to form a composite application.

In this chapter we'll explain how to declare a Java component in an SCA composite file, how to write a Java class and interface with SCA annotations, and how to map a Java class and interface into an SCA component type. We'll describe the Java language–specific details of the interaction patterns we discussed generally in chapter 4, and we'll finish this chapter with some discussion of error handling.

You're probably somewhat familiar with the Java implementation type because we've already used it in earlier chapters in this book. Here we'll be bringing together all the Java language–specific information in one place, so you'll see some things in this chapter that we've already said. Because there's a lot of information, you may want to skip some of the more advanced sections the first time through. For example, the callback and conversational patterns have been covered at a high level in chapter 4. Until you need to know the details, you can safely skip over them.

The samples we'll use in this chapter are primarily based on the payment-related components from the full travel-booking application. The payment components deal with payment processing once the user has decided to buy the trips held in the shopping cart. You haven't come across these components yet because they're not included in the introductory version of the travel-booking application in chapter 1. You'll get to know them in some detail in this chapter and the next. Let's start by reviewing the basics of how you'll define a Java language–based component implementation.

5.1 Defining a Java component implementation

Defining a Java component implementation is simple. We'll use the Payment component from the TuscanySCATours application as an example. In its simplest form the Payment component is defined using the implementation.java element, which references the Java class `com.tuscanyscatours.payment.impl.PaymentImpl`. The following snippet, from the composite file in the `payment-java` sample contribution, shows how the component is defined and made available in the Payment composite:

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com"
           name="payment">

  <component name="Payment">
```

```
    <implementation.java
                    class="com.tuscanyscatours.payment.impl.PaymentImpl" />
  </component>

</composite>
```

Figure 5.1 shows how the Payment component Java implementation fits, in relation to SCA services, references, and properties.

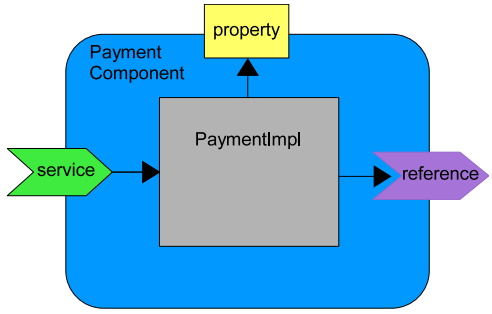


Figure 5.1 The Java implementation (PaymentImpl) provides the business logic for a component, providing services and using references and properties.

In the TuscanySCATours application, the Payment component collaborates with other components. Figure 5.2 shows a composite application that consists of the Payment component connected to other components, each implemented using implementation.java.

The CustomerRegistry component looks up customer payment information based on the customer's ID, the CreditCardPayment component handles the payment itself, and the EmailGateway component notifies the customer of the payment status.

We'll use this example throughout this chapter and the next to demonstrate how the various implementation types work. The TuscanySCATours application demonstrates the Java language version of the Payment component, along with the CustomerRegistry and EmailGateway components, in the `payment-java` contribution. The CreditCardPayment component is described in the `creditcard-payment-jaxb` contribution. Not all of the features of the Java implementation type have matching sample code, but this chapter includes code snippets based on the sample Payment component.

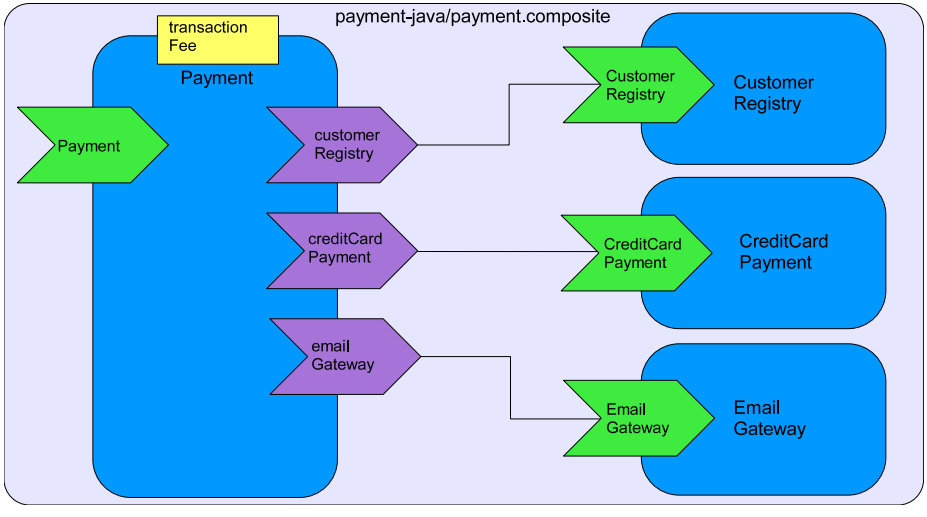


Figure 5.2 The Payment Java component connected to the other components that it depends on

The following code snippet shows the configuration of the Payment component as it appears in a composite file:

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com"
           name="payment">

  <component name="Payment">
      <implementation.java
                   class="com.tuscanyscatours.payment.impl.PaymentImpl" />
      <service name="Payment">
          <binding.ws uri="http://localhost:8081/Payment" />
          <binding.sca />
      </service>
      <reference name="customerRegistry" target="CustomerRegistry"/>
      <reference name="creditCardPayment" target="CreditCardPayment">
          <binding.ws uri="http://localhost:8082/CreditCardPayment" />
      </reference>
      <reference name="emailGateway" target="EmailGateway"/>
      <property name="transactionFee”>0.02</property>
  </component>
    ...

</composite>
```

Let's move on and use the payment example to look at how Java annotations can be added to a Java class to define SCA services, references, and properties for the Payment component. You can use different styles to do this, and we'll look at these styles in the following sections.

5.2 Using SCA annotations in Java implementations

A component type describes the shape of a component in terms of the services it provides and the references and properties that it uses. SCA defines several Java annotations that allow the component implementation developer to describe the component type easily. The annotations are defined in the SCA Java Common Annotations and APIs specification (http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf).

In our example, the `PaymentImpl` class implements the `Payment` interface and provides the business logic for the Payment component. The `Payment` interface is generated from `Payment.wsdl`, using code in the `pom.xml` or `build.xml` files, and can be found in the payment-java/target/jaxws-source directory. Listing 5.1 shows the source code for `PaymentImpl`. The annotations `@Service`, `@Reference`, and `@Property` are used to identify services, references, and properties. The fields associated with `@Reference` and `@Property` annotations will be initialized automatically by the Tuscany runtime.

Listing 5.1 Implementation class for the Payment component

```
@Service(Payment.class)
public class PaymentImpl implements Payment {
    @Reference
    protected CustomerRegistry customerRegistry;
```

```
    @Reference
    protected CreditCardPayment creditCardPayment;

    @Reference
    protected EmailGateway emailGateway;

    @Property
    protected float transactionFee = 0.01f;

    public String makePaymentMember(String customerId, float amount) {
      Customer customer = customerRegistry.getCustomer(customerId);
      String status = creditCardPayment.authorize(customer.getCreditCard(),
                                                  amount + transactionFee);
      emailGateway.sendEmail("order@tuscanyscatours.com",
                             customer.getEmail(),
                             "Status for your payment",
                             customer + " >>> Status = " + status);
      return status;
    }
}
```

Figure 5.3 gives a pictorial representation of how the `PaymentImpl` class uses SCA Java annotations to define SCA services, references, and properties. In this figure, `@Service` is used to describe Payment as an available service. The `@Reference` annotation is used to describe the dependency of the Payment component on the CreditCardPayment service. The `@Property` annotation is used to describe how property values in the business logic can be set from the composite file. The rest of the `PaymentImpl` class contains business logic to implement the business process.

You now have the basic idea of how a Java class can be used to implement an SCA component. One thing you may have noticed is that services and references in the

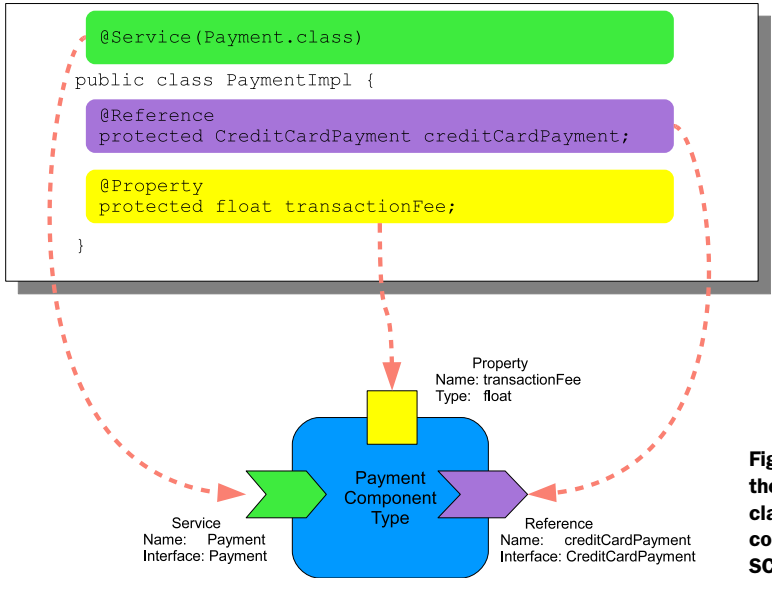


Figure 5.3 Mapping the Java implementation class to an SCA component type using SCA annotations

Java implementation are described using Java interfaces. Let's look in more detail at how Java interfaces are used.

5.3 Services and references with Java interfaces

Each SCA component has a formal contract, called the component type, that defines its service, references, and properties. Tuscany can derive the component type directly from the component's Java implementation rather than having to specify it manually.

In this section we'll look at the role of Java interfaces in defining the component type, and we'll start by looking at the difference between local and remote services.

5.3.1 Identifying local and remote services

A composite application can consist of components that run in the same JVM or that communicate remotely across the network. You may remember from chapter 4 that SCA allows component service interfaces to be defined as local or remote. A Java interface is local by default and describes a component that must be running within the same JVM as the reference that calls it. The following snippet shows a local Java interface:

```
public interface CustomerRegistry {
    Customer getCustomer(String id);
}
```

Remotable interfaces can also be used for components running in the same JVM, but they're flexible enough to be distributed over the network too. The data exchange semantics for remotable interfaces is pass-by-value. You can annotate an interface with `@Remotable` to denote that it's a remotable interface. The following snippet shows a remotable Java interface:

```
@Remotable
public interface EmailGateway {
    public boolean sendEmail(String sender,
                             String recipient,
                             String subject,
                             String body);
}
```

In addition to Java interfaces that are annotated with `@Remotable`, the Java language itself defines certain patterns for interfaces that are considered remotable, for example, the `javax.rmi.Remote` interface in RMI.

Any object that's a remote RMI object must directly or indirectly implement this interface. Another example is a Java interface that's annotated with the JAX-WS annotation

RMI

The Java Remote Method Invocation API (or RMI) is an API that allows Java applications to invoke operations on objects in another JVM that have been exposed as RMI services. RMI is provided as a core part of the Java language.

`@javax.jws.WebService` to indicate that the interface defines a Web Services interface that's remotable.

JAX-WS

JAX-WS stands for Java API for XML Web Services. It defines rules and Java annotations for mapping Web Services to Java classes and vice versa. You can read the specification at http://jcp.org/en/jsr/detail?id=224. You don't need to be an expert in JAX-WS or associated technologies such as JAXB and the various WS-* standards in order to use web services with Tuscany. However, gaining some background knowledge from the many online sources will help you solve any problems that arise as you build and deploy applications. In the TuscanySCATours sample we'll mainly use JAX-WS for generating Java interfaces from WSDL files. For example, if you look at the contributions/payment-java/pom.xml file, you can see the `jaxws-maven-plugin` configured to generate Java classes based on the Payment.wsdl and CreditCardPayment.wsdl files. If you prefer to use Ant, the build.xml file uses a wsimport target that's been built for the sample and can be found in the antdefs.xml file in the sample's root directory. JAX-WS is built directly into the Java 6 JDK but not the Java 5 JDK. If you're using Java 5, then the TuscanySCATours sample ships with the JAX-WS JARs in the lib/jaxws directory, and the various build scripts will look there if required.

In SCA, we automatically treat the Java interfaces that follow such protocol-specific remote patterns as remotable interfaces. For example, if you already have a JAX-WS–style interface, such as that shown in the following listing, then the Tuscany runtime will treat it as a remotable SCA service interface.

Listing 5.2 A remotable interface generated from a WSDL by JAX-WS: payment.Payment

```
@WebService(name = "Payment",
            targetNamespace = "http://www.tuscanyscatours.com/Payment/")
@XmlSeeAlso({ObjectFactory.class})
public interface Payment {
  @WebMethod(action = "http://www.tuscanyscatours.com/Payment/makePayment")
  @WebResult(name = "Status", targetNamespace = "")
  @RequestWrapper(localName = "makePaymentMember",
                  targetNamespace = "http://www.tuscanyscatours.com/Payment/",
                  className = "payment.MakePaymentMemberType")
  @ResponseWrapper(localName = "makePaymentMemberResponse",
                  targetNamespace = "http://www.tuscanyscatours.com/Payment/",
                  className = "payment.MakePaymentMemberResponseType")
  public String makePaymentMember(
                       @WebParam(name = "CustomerId", targetNamespace = "")
                       String customerId,
                       @WebParam(name = "Amount", targetNamespace = "")
                       float amount);
}
```

This looks a little complicated but will be familiar to those used to Java interfaces generated according to JAX-WS rules. The point here is that you don't have to use SCA annotations to make a Java interface remotable.

It's also important to understand that remotable interfaces don't mean that the components have to be deployed remotely. They can run just as well within a local composite assembly. We often start by testing with a local assembly of components that are then deployed to a distributed environment later on.

5.3.2 Implicit and explicit definition of component interfaces

When we show examples of component definitions from composite files, we usually show the form that assumes that a component's service and reference interfaces are derived implicitly from the implementation. For example, let's assume our Payment component is being provided over Web Services and, in turn, accesses other services over Web Services.

```
<component name="Payment">
    <implementation.java
                  class="com.tuscanyscatours.payment.impl.PaymentImpl" />
    <service name="Payment">
        <binding.ws/>
    </service>

    <reference name="creditCardPayment">
        <binding.ws uri="http://localhost:8082/CreditCardPayment"/>
    </reference>
    ...
</component>
```

The Java class `PaymentImpl` implements the Java interface `Payment`, and that, in turn, describes the component service interface. The Java type of the reference field creditCardPayment describes the component reference interface in this example. The default behavior is equivalent to the following composite definition:

```
<component name="Payment">
      <implementation.java
                  class="com.tuscanyscatours.payment.impl.PaymentImpl" />
      <service name="Payment">
         <interface.java interface="com.tuscanyscatours.payment.Payment" />
         <binding.ws/>
      </service>
      <reference name="creditCardPayment">
         <interface.java interface=
              "com.tuscanyscatours.payment.creditcard.CreditCardPayment" />
          <binding.ws />
      </reference>
  </component>
```

It's sometimes the case that the technology used to describe the component implementation's interfaces is different from that used to describe the SCA component's service and reference interfaces. Imagine that a WSDL description has been generated

by hand, or by a tool such as Eclipse, to describe the Payment component service interface. This WSDL can be referenced explicitly from the composite file as follows:

```
<service name="Payment">
  <interface.wsdl interface="http://.../Payment/#wsdl.interface(Payment)"/>
  <binding.ws/>
</service>
```

> **Mapping between Java and WSDL interfaces**
>
> WSDL provides an XML-centric description of a service interface and the operations it contains. The data that passes into and out of the service operations is represented using XML types. Operations in a Java interface can be mapped to operations in a WSDL interface according to the rules described in the JAX-WS specification (http://jcp.org/en/jsr/detail?id=224). This, in turn, implies the use of the JAXB (http://jcp.org/en/jsr/detail?id=222) rules for mapping Java objects that appear in operation signatures to XML schema. At its simplest, primitive Java types map to XML schema simple types while complex Java types, JavaBeans or collections, map to XLM schema complex types. There are complexities in the mapping, however, depending on what style of WSDL you choose. Many good online resources discuss the topic of Java-to-WSDL mapping. Also, chapter 9 describes in more detail the runtime transformations Tuscany can perform between WSDL-defined XML types and Java.

Internally the Tuscany runtime holds a custom model of a service's interface that's neither WSDL nor a Java interface. It's able to convert both Java interfaces and WSDL descriptions to this internal format and check that the interfaces are compatible. When converting between Java interfaces and WSDL service interfaces, the Tuscany runtime applies the default Java interface and WSDL mapping rules described by JAX-WS.

5.3.3 Interface compatibility and mapping

For a component service and implementation interface to be compatible, the component service interface must describe a compatible subset of the operations that the component implementation interface provides. This means that an implementation may provide some operations that the SCA service doesn't make available for use by other components. For example, if our Payment component implementation interface exposes operations for handling payments from both members and nonmembers, we may want to configure our component's service interface so that only the operation that accepts payments from members is exposed. Continuing with our WSDL example, let's assume we've built a WSDL description with only a `paymentMember` operation. This WSDL can be referenced from a `interface.wsdl` element added to the Payment service element of the Payment component description, as we did in the previous section. The Payment service will then accept only calls to the `paymentMember` operation.

The same approach holds for references. References within the Java implementation will be described using Java interfaces. But the SCA reference could be described using WSDL. For example, a WSDL file may have been created by the credit card payment processing company in order to describe the remote services they provide. They may have generated the WSDL by hand or used a WSDL generation tool. The WSDL can be used to configure the component reference that will connect to the CreditCardPayment service.

```
<reference name="creditCardPayment">
    <interface.wsdl interface=
           "http://.../CreditCardPayment/#wsdl.interface(CreditCardPayment)"/>
    <binding.ws />
</reference>
```

Again, the Java interface and the WSDL interface are both mapped to Tuscany's interface representation, and the runtime checks that the component reference interface describes a compatible superset of the operations described by the implementation reference interface. To put it another way, the remote service may provide some operations that the implementation never calls. If the interfaces don't match, the Tuscany runtime will raise an error when the application is started.

5.3.4 Transforming messages between interfaces

A Java implementation may use any number of technologies to describe the data that flows in and out of the interface operations that the implementation supports. For example, you could build a Java component implementation that uses JAXB, SDO, or POJOs to describe interface operation parameters and return types.

> **Data formats for parameters and return types**
>
> There are many different ways to represent structured data in the Java language. You could use a plain old Java object (POJO). Alternatively, you could use a technology designed to represent the data and map it to XML. Java Architecture for XML Binding (JAXB) and Service Data Objects (SDO) are examples of this kind of technology.

Messages passing between components often need to be converted from one form to another in order to be passed out to or received from the network by an SCA binding.

Fortunately, Tuscany defines a databinding framework that takes care of this data format transformation. In this way, the Java implementation deals entirely with familiar data objects and is insulated from any data formats that are required by the Tuscany bindings. The databinding framework is described in chapter 9. Here we'll focus on how data is handled within the Java implementation type.

5.3.5 Pass-by-reference and pass-by-value

By default, data is exchanged between remotable service interfaces using pass-by-value semantics. Pass-by-value means that input data is copied between reference and service, and any changes that a service makes to the input data are not automatically seen by the referencing component. Changes will be seen by the referencing component only if the service explicitly returns the changed data as output data. To make this work, the Tuscany infrastructure has to ensure that the data is copied between reference and service.

In some cases, the called component implementation knows it won't change the request data, and the client is free to change the response data. To allow the runtime to optimize data exchanges, you can annotate the implementation class or methods with `@AllowsPassByReference`. This indicates to the runtime that the data can be safely passed by reference over the remotable interface, for example:

```
@AllowsPassByReference
public class PaymentImpl {

      public boolean makePayment(Customer customer, float charge) {
         ...
      }
}
```

The runtime is able to use this information to optimize communications when two connected components are running in the same JVM. When communicating remotely from one JVM to another, communications will always be pass-by-value.

Now that you know how Java interfaces are exploited generally, let's look in detail at how the services that an SCA component provides are defined.

5.4 Java component services

The Java implementation class for an SCA component can provide one or more SCA services. In this section, we'll describe how SCA services can be declared in the implementation class, how to define an interface for the service, and how to derive the name of the service from its interface.

5.4.1 The @Service annotation

The `@Service` annotation can be used with a Java class to explicitly define the SCA services provided by a component. The following snippet defines an SCA service whose interface is `CustomerRegistry`:

```
@Service(CustomerRegistry.class)
public class CustomerRegistryImpl implements CustomerRegistry {
...
}
```

If there's more than one service, the interface's parameter for `@Service` can be used to list all the services by including each interface as an element in an array.

```
@Service(interfaces={CustomerRegistry.class, AnotherService.class})
public class CustomerRegistryImpl implements CustomerRegistry,
                                              AnotherService {
    ...
}
```

The `@Service` annotation isn't the only way to identify services for a Java component. Let's look at the alternatives.

5.4.2 Alternatives to the @Service annotation

When the `@Service` annotation isn't present, the list of interfaces implemented by the Java class will be introspected by Tuscany to find out if there are any remotable interfaces. Each remotable interface will become an SCA service described by that interface. You've seen previously that the `Payment` interface is a remotable interface. Here `Payment` is recognized as a service because it's implemented by the `PaymentImpl` class and is remotable:

```
public class PaymentImpl implements Payment, ... {
    ...
}
```

If no remotable interfaces are found, or the class doesn't implement any interfaces, then a single SCA service is assumed whose interface, and consequently the service name, is the implementation class. In the following snippet, the SCA service name is CustomerRegistryImpl (not CustomerRegistry) and all of the methods in the `CustomerRegistryImpl` class are available as operations on the service interface. This is because, although `CustomerRegistryImpl` does implement the `CustomerRegistry` interface, the interface isn't remotable.

```
public class CustomerRegistryImpl implements CustomerRegistry {
    ...
}
```

> **Service naming gotcha**
>
> This often catches the new user. You're happily creating components that implement remotable Java interfaces. You naturally, and correctly, use the name of the interface as the name of the component's service. Then you implement a component using a nonremotable interface, and all of a sudden the runtime reports errors. In this case, the service name is the name of the implementation class and not the interface.

For Java components without an `@Service` annotation, the name of the service is formed from the local (last) part of the interface name. For example, if the interface is `com.tuscanyscatours.payment.Payment`, then the service name is Payment.

Table 5.1 gives a summary of the various ways to declare SCA services in Java implementations.

Table 5.1 Ways to declare SCA services in Java implementations

Implementation class	Service name

`@Service(Payment.class)`
`public class PaymentImpl`
`  implements Payment {`
`  ...`
`}`
	Payment

`public class PaymentImpl`
`  implements Payment {`
`  ...`
`}`
	Payment
	(`Payment` interface is remotable.)

`@Service(Payment.class)`
`public class PaymentImpl {`
`  ...`
`}`
	Payment

`@Service(CustomerRegistry.class)`
`public class CustomerRegistryImpl`
`  implements CustomerRegistry {`
`  ...`
`}`
	CustomerRegistry

`public class CustomerRegistryImpl`
`  implements CustomerRegistry {`
`  ...`
`}`
	CustomerRegistryImpl
	(`CustomerRegistry` interface is local.)

Now that you've seen the various ways that the Java implementation can be mapped to SCA component services, let's look at how SCA component references are handled.

5.5 Java component references

A Java component often depends on other services. We'll need to declare SCA references in order to exploit these other services. An SCA reference has the following attributes:

- `name`—The name that uniquely identifies the reference in the component
- `multiplicity`—Controls how many target services can be associated with the reference
- `interface`—The interface of the reference
- `target`—One or more services that the reference will be wired to

Reference targets are configured in the composite file rather than from within the component implementation. We discussed this already in chapter 2, and the advantage is

that the target can be changed without having to change the component implementation code. You identify references in the Java implementation, and Tuscany ensures that each reference proxy is connected to the correct service.

The interface of a reference has to be compatible with the service to which it's connected. We already discussed interfaces in section 5.3. With the Java implementation, the reference interface type can be introspected from the implementation. To do this, the Tuscany runtime looks for those fields annotated with `@Reference`. Let's start by looking at how this annotation works.

5.5.1 The @Reference annotation and reference injection

A referenced service needs to be found before it can be called from within a referencing component's implementation. The Tuscany runtime injects the proxy for each reference service when it detects the `@Reference` annotation. The `@Reference` annotation also allows the Tuscany runtime to determine the reference interface type and the type of proxy required. The `@Reference` annotations can be used in three different styles. You can mix the following styles in your component implementation, and in all three cases the reference interface here is `CreditCardPayment`.

1. *A public or protected field*—The service proxy will be injected directly into the field.

```
@Reference
protected CreditCardPayment creditCardPayment;
```

2. *A public or protected setter method*—The service proxy will be injected by calling the setter method.

```
@Reference
public void setCreditCardPayment(CreditCardPayment creditCardPayment) {
    ...
}
```

3. *A parameter on the class constructor*—The service proxy will be injected as a parameter when the object is instantiated using the constructor.

```
@Constructor({"creditCardPayment", ...})
public PaymentImpl(@Reference CreditCardPayment creditCardPayment, ...) {
    ...
}
```

Style 1 allows the injection of the service proxy without other logic. Both style 2 and style 3 allow code to perform additional checking when the service proxy is injected.

In style 2, the service proxy can be reinjected via the setter method after a component implementation has been created, whereas styles 1 and 3 make the SCA reference immutable.

The Tuscany runtime doesn't currently take advantage of the ability to reinject references because the domain implementation is static and references aren't changed

once a component instance is created. The intention is to provide more dynamic domain support in the future.

5.5.2 Reference naming

Each reference has a unique name in the component. The name can be explicitly specified by the `name` attribute of the `@Reference` annotation, for example:

```
@Reference(name="creditCardPayment")
protected CreditCardPayment ccPayment;
```

This explicitly declares an SCA reference named `creditCardPayment`. If the `name` attribute isn't present, then Tuscany derives the reference name from the name of the field or method where `@Reference` is declared.

```
@Reference
protected CreditCardPayment creditCardPayment;
```

The reference name is again `creditCardPayment` and is derived from the name of the `creditCardPayment` field. If a setter method is marked as a reference but without an explicit reference name, the reference name is derived from the available information.

```
@Reference
public void setCreditCardPayment(CreditCardPayment creditCard);
```

Again, the reference name is `creditCardPayment`. This is the property name for the setter method as defined by the JavaBeans specification. The `set` prefix is trimmed from the method name, and the first letter is converted to lowercase unless the second letter is uppercase too. Here `C` becomes `c`.

5.5.3 Reference multiplicity

Each SCA reference can potentially be wired to any number of target services. This is controlled by the multiplicity attribute on the reference element in the composite file:

- Exactly one service provider (multiplicity=`1..1`)
- An optional service provider (multiplicity=`0..1`)
- At least one service provider (multiplicity=`1..n`)
- Any number of service providers (multiplicity=`0..n`)

When using implementation.java, the lower bound of the SCA reference multiplicity is determined by the `required` attribute of the `@Reference` annotation, for example:

```
@Reference(required=true)
```

If `required` is `true`, then the lower bound of the multiplicity is `1`; else it's `0`. If the `required` attribute isn't specified, the `required` value defaults to `true`.

The upper bound is controlled by the Java type. If the Java type is a subclass of `java.util.Collection` or an array, the upper bound is `n`; otherwise the upper bound is `1`.

Table 5.2 summarizes the range of reference multiplicities that can be described.

Table 5.2 The mapping between `@Reference` and reference multiplicity

`@Reference` declaration	SCA reference	Multiplicity

`@Reference(required=false)`
`protected OtherService`
`optionalService;`
	Name:
	`optionalService`
	Interface:
	`OtherService`
		0..1

`@Reference(required=true)`
`protected OtherService`
`requiredService;`
or
`@Reference`
`protected OtherService`
`requiredService;`
	Name:
	`requiredService`
	Interface:
	`OtherService`
		1..1

`@Reference(required=false)`
`protected Collection<OtherService>`
`optionalServices;`
or
`@Reference(required=false)`
`protected OtherService[]`
`optionalServices;`
	Name:
	`optionalServices`
	Interface:
	`OtherService`
		0..n

`@Reference(required=true)`
`protected Collection<OtherService>`
`requiredServices;`
or
`@Reference(required=true)`
`protected OtherService[]`
`requiredServices;`
	Name:
	`requiredServices`
	Interface:
	`OtherService`
		1..n

You now know how to declare references in the Java implementation class. Let's move on and look at properties.

5.6 Java component properties

A flexible business service needs to adapt to business changes. For example, a catalog can list items using different currencies, or a discount rate can be changed based on prevailing business conditions. SCA allows business components to present configurable attributes as properties. An SCA property has the following attributes:

- `name`—The name of a property that uniquely identifies the property in the component.
- `type`—The type for the property.
- `many`—Controls whether multiple values can be supplied for the property.
- `mustSupply`—Indicates whether a value has to be supplied.

- `value`—The value of the property that's typically configured in the composite file. It can be changed in the composite file without modifying the Java class.

The property `name`, `type`, `many`, and `mustSupply` values are usually defined in the component implementation, whereas the property `value` is supplied in the composite file, for example:

```
<component name="Payment">
    <implementation.java
                    class="com.tuscanyscatours.payment.impl.PaymentImpl" />
        ...
    <property name="transactionFee">0.02</property>
</component>
```

This means that property values can be set at the time the composite files are created, and there's no need to change a component's implementation in order to set property values.

In the following subsections we'll show how properties are identified, named, and typed and also look at how property value multiplicity is handled. We'll start by looking at identifying properties in Java component implementations using the `@Property` annotation.

5.6.1 The @Property annotation and property injection

As with references, SCA and Tuscany provide various methods for declaring a property in the Java implementation class:

1. *A public or protected field*—The Tuscany runtime will inject the value of the property that's specified in the composite into the component implementation.

   ```
   @Property
   protected float transactionFee;
   ```

2. *A public or protected setter method*—The Tuscany runtime will inject the property value by calling the setter method.

   ```
   @Property
   public void setTransactionFee(float transactionFee) {
       ...
   }
   ```

3. *A parameter on the class constructor*—The service proxy will be injected as a parameter when the object is instantiated using the constructor.

   ```
   @Constructor({"transactionFee", ...})
   public PaymentImpl(@Property float transactionFee, ...) {
       ...
   }
   ```

As with references, you can mix these styles within a component implementation. Style 1 allows properties to be injected without the implementation developer needing to write any extra code. Styles 2 and 3 allow the implementation developer to add additional code to the injection process.

Styles 1 and 3 inject immutable properties, because they're injected once when the component instance is created. The point at which a component instance is created depends on the scope of the component, for example, a `STATELESS` scoped component instance is created each time a message arrives. Component scope is discussed later in section 5.7. Style 2, in theory, allows property values to be reinjected while the component instance is running, although the Tuscany runtime currently doesn't exploit this feature.

5.6.2 Property naming

Each property has a unique name in the component. The name can be explicitly specified by the `name` attribute of the `@Property` annotation, for example:

```
@Property(name="transactionFee")
protected float txFee;
```

This declares an SCA property named `transactionFee`. If the `name` attribute isn't present, then the name is derived from the field or method where `@Property` is declared.

```
@Property
protected float transactionFee;
```

Again the property name is `transactionFee`, which is the name of the transactionFee field. If a setter method is marked as a property but without an explicit property name, the property name is derived from the available information.

```
@Property
public void setTransactionFee (float transactionFee);
```

The property name is again `transactionFee` based on the property name for the setter method as defined by the JavaBeans spec. The `set` prefix is trimmed from the method name, and the first letter is converted to lowercase. Here `T` becomes `t`.

5.6.3 Property types

The Java type of a property has to be able to map into an XML type. The precise details of this mapping depend on which mapping is in force for a particular component property. By default Tuscany assumes a JAXB mapping, and you can find the details of this in the JAXB specification (http://jcp.org/en/jsr/detail?id=222). In general, simple types, such as `int`, `float`, or `String`, can be directly mapped into simple XML types. Our payment example has already shown a `float` type being specified in a Java field and being set in the composite file.

Complex XML types will map to a JavaBean. We described in chapter 2 how complex XML structures can be used to set properties in the composite file, so we won't repeat the details here. It's worth noting, though, that in a Java implementation you can represent complex typed properties using any of the Java object technologies for which Tuscany supports an XML-to-Java transformation, such as JAXB or SDO. Chapter 9 discusses the databinding layer and how these transformations work.

5.6.4 Property value multiplicity

A property can be single valued or many valued. A many-valued property is represented in the Java component implementation using a collection class. A property can also be required or optional. This is controlled by the `mustSupply` attribute in the composite file. In a Java implementation, the `required` attribute of the `@Property` annotation specifies whether `mustSupply` is set or not. Table 5.3 shows the various forms of the `@Property` annotation and the corresponding property multiplicity.

Table 5.3 The mapping between `@Property` and SCA component property

`@Property` declaration	`mustSupply`	SCA component property	Many valued

`@Property(required=false)`
`protected String`
`optionalProperty;`
or
`@Property`
`protected String`
`optionalProperty;`
	`false`	Name:
		`optionalProperty`
		Type:
		`String`
			`false`

`@Property(required=true)`
`protected String`
`requiredProperty;`
	`true`	Name:
		`requiredProperty`
		Type:
		`String`
			`false`

`@Property(required=false)`
`protected Collection<String>`
`optionalProperties;`
or
`@Property(required=false)`
`protected String[]`
`optionalProperties;`
	`false`	Name:
		`optionalProperties`
		Type:
		`String`
			`true`

`@Property(required=true)`
`protected Collection<String>`
`requiredProperties;`
or
`@Property(required=true)`
`protected String[]`
`requiredProperties;`
	`true`	Name:
		`requiredProperties`
		Type:
		`String`
			`true`

To configure a many-valued property in the composite file, you'll need to set the `many` attribute to `true` and provide multiple whitespace-separated values inside the `property` element, for example:

```
<property name="validCurrencies" many="true" type="xsd:string">
   "USD" "GBP" "EUR"
</property>
```

Note that the set of whitespace characters used for separating property values includes spaces, tabs, line feeds, and carriage returns but doesn't include commas.

A property with a complex XML type can be many valued as well. It's a matter of including multiple complex values in the `property` element.

So far in the implementation.java section, you've learned how to declare SCA services, references, and properties using SCA annotations. Now we'll look at how to control the creation of the component instances that use these constructs.

5.7 Java component instance creation and scope

The business logic for a Java component may have to manage its state. State management is directly related to how component instances are created and controlled. SCA defines a number of different component scopes that control when new instances of components are created and when they're removed.

In the following subsections we'll show how to control the scope of a component instance and how to execute code when a component instance is created and destroyed. Let's start by looking at how to specify a component's scope using the `@Scope` annotation.

5.7.1 Stateless, composite, and conversational scopes

To help Java language developers manage implementation states, SCA introduces the implementation scope concept. In the Java language you can annotate the implementation class with `@Scope` to tell the SCA runtime how to manage instances of a component's Java implementation class. Note that other application frameworks also define @Scope annotations, for example, the Spring framework. It you're using Tuscany in combination with such technologies, you'll need to look for potential conflicts and qualify the Tuscany Scope annotation with its full namespace, `org.osoa.sca.annotations.Scope`, as required. Table 5.4 shows the different scopes that can be used in the Tuscany runtime.

Table 5.4 The implementation scopes supported by the Tuscany runtime

Scope	Description
`STATELESS`	Create a new component instance on each call.
`COMPOSITE`	Create a single component instance for all calls.
`CONVERSATION`	Create a component instance for each conversation.

Using the `STATELESS` scope, a different implementation instance will be created to service each request. Implementation instances will be newly created automatically by the Tuscany runtime each time the component is called. We can imagine a future enhancement of the Tuscany runtime where instances are drawn from a pool of instances for performance reasons, but Tuscany doesn't do this at the moment. A stateless Java component is usually good for scalability because there's no overhead required for state maintenance.

For example, the CreditCardPayment component receives all of the data it needs to complete its processing as input parameters and doesn't need to maintain any state from one call to the next.

```
@Scope("STATELESS")
public class CreditCardPaymentImpl implements CreditCardPayment {
    ...
}
```

The `CreditCardPaymentImpl` class will be guaranteed by the SCA runtime to be thread safe because concurrent requests from different threads won't be dispatched to the same object instance. To put it another way, a new component instance is created for each thread.

The CustomerRegistry component has a different requirement. It needs to access a database to work with customer data. To ensure data consistency and leverage caching, it's desirable that a single implementation instance be used to serve all the requests in the containing composite.

```
@Scope("COMPOSITE")
public class CustomerRegistryImpl implements CustomerRegistry {
    ...
}
```

Note that the implementation of a `COMPOSITE`-scoped component must be designed to be thread safe because all requests to the component will go through the same component instance.

If you need a `COMPOSITE`-scoped implementation instance to be created when its containing composite is started, that is, when the Tuscany node that's running it is started, you can annotate the class with `@EagerInit`. For example, connecting to a database and loading some data into memory might be slow. You can immediately perform database connection processing when the component instance is created and before any requests arrive.

The third scope is `CONVERSATION`. Conversational interactions were described in chapter 4 and represent a series of correlated interactions between a client and a target service. The same implementation instance of a `CONVERSATION`-scoped component will be used throughout the lifetime of a conversation.

You may be wondering if there's a way you can do some processing whenever a Java component instance is created or destroyed. Let's look at that next.

5.7.2 Interacting with component instance creation and destruction

The SCA runtime allows special processing to happen when a scope begins and ends. To act on these lifecycle events, the application code can denote two special methods with `@Init` and `@Destroy`.

```
@Service(CustomerRegistry.class)
@Scope("COMPOSITE")
@EagerInit
public class CustomerRegistryImpl implements CustomerRegistry {
```

```
    @Init
    public void init() {
        ...
    }

    @Destroy
    public void destroy() {
        ...
    }
}
```

In this code snippet, the `init()` operation will be called after the `CustomerRegistryImpl` class is instantiated. We could have code here to connect to the customer database and prefetch some information into a cache. In the `destroy()` operation, we can then dispose of the cache so that memory is freed up.

We've now looked at the basics of constructing a Java component implementation. Now we'll look at how the implementation code interacts with the services that references are wired to. Chapter 4 describes a number of interaction patterns, and we'll discuss how both callbacks and conversations are controlled using the APIs available to a Java implementation. Let's start by looking at callbacks.

5.8 Making callbacks

In chapter 4 you saw how the callback interaction pattern can be used by a component to make callbacks to a client that has invoked a service of the component. If you don't need to know the details of how to use callbacks with Java components at the moment, you can safely skip forward to section 5.9.

For the Java implementation type, a service makes a callback using a *callback proxy*. Callback proxies can be used by service implementations to make service calls in a similar way to the reference proxies we already described. The important difference is that a callback proxy's destination is selected at runtime based on a previous call to the service, whereas a reference proxy's destination is selected once at node start time based on reference wiring and binding configuration.

To further illustrate how callbacks can be used, we'll add a callback to our Payment example. The CreditCardPayment service might need to perform additional security checks to authorize certain payments. For example, for large transactions, the credit card company might need to request an additional security password from the customer before approving the transaction. We can use an SCA callback for this password request.

In the following sections we'll use this Payment callback example to explore various aspects of the callback programming model that are specific to Java implementations. We'll show how callbacks are declared using bidirectional interfaces, and we'll review how bidirectional services obtain callback proxies by injection. We'll look at how bidirectional services can obtain callback proxies using API calls instead of by injection, and we'll demonstrate how bidirectional services can use callable references to provide more flexibility in making the callback. We'll also show how clients can identify and correlate individual callbacks using callback IDs. Finally, we'll look at how clients can delegate callback handling to another service.

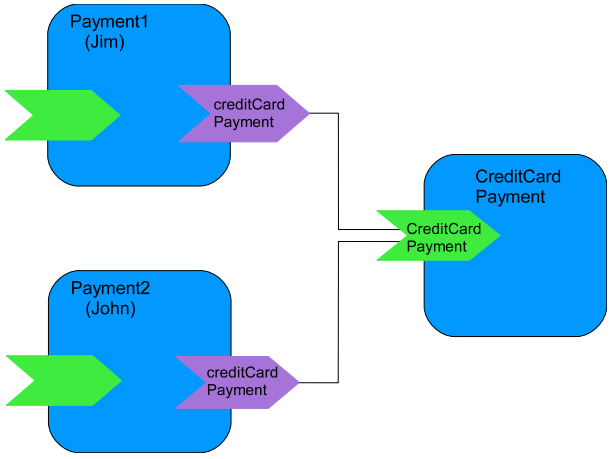


Figure 5.4 Two different payment components are wired to the same bidirectional service of a credit card payment component.

5.8.1 The credit card security callback scenario

We'll use the credit card security password example to illustrate the various steps involved in creating implementation instances, injecting callback proxies, and making callbacks. You can find this example in the `payment-java-callback` sample contribution. Figure 5.4 shows two payment components calling the same credit card payment service on behalf of different customers.

The interaction diagram in figure 5.5 uses the components shown in figure 5.4 and shows Jim making a payment of $1500, whereas John is making a payment of $500.

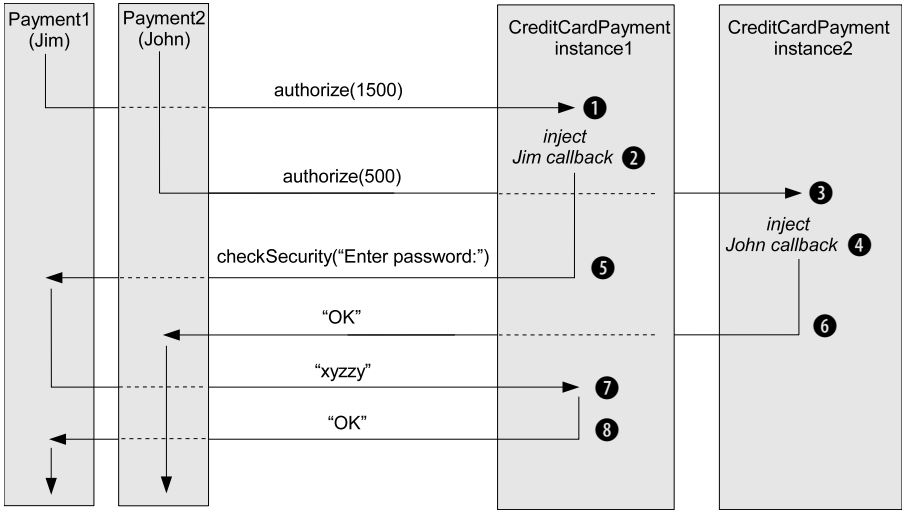


Figure 5.5 Two different client components are processing payments for Jim and John using the same CreditCardPayment component. For each payment authorization request, a different instance of the CreditCardPayment component is created. Each instance of CreditCardPayment gets the appropriate callback proxy injected and can use this to make callbacks to the correct client component.

In figure 5.5, the first interaction is a call to the credit card payment service to authorize Jim's payment of $1500 ❶, causing Tuscany to create a new instance of CreditCardPayment and inject this instance with a callback proxy for Jim ❷. Next, a call for John's payment of $500 is made ❸, causing another instance of CreditCardPayment to be created and injected with a callback proxy for John ❹. Jim's payment is larger and needs an additional security check, so the CreditCardPayment implementation calls back to Jim requesting a password ❺. John's payment is below the limit for additional security, so the CreditCardPayment implementation authorizes this payment without making a callback ❻. Jim's callback responds with the correct password ❼, and the CreditCardPayment implementation authorizes Jim's payment and returns ❽.

We'll use this scenario to look at how bidirectional services and callback clients are defined and programmed with the Java implementation type.

5.8.2 Creating a bidirectional interface with the @Callback annotation

As we described in chapter 4, the first step in implementing a callback with a Java interface is to define a bidirectional service interface with a callback interface. The following code snippet gives a quick reminder of how to do this:

```
@Remotable
public interface CreditCardSecurity {
    String checkSecurity(String securityPrompt);
}

@Remotable
@Callback(CreditCardSecurity.class)
public interface CreditCardPayment {
    String authorize(CreditCardDetailsType ccDetails, float amount);
}
```

In this code we've defined an interface, `CreditCardSecurity`, and used the `@Callback` annotation to add this as the callback interface for the `CreditCardPayment` interface. Now we'll look at how the Java component implementation makes a callback.

5.8.3 The service programming model for callbacks

To make a callback, the service implementation needs a callback proxy. As with reference proxies, Tuscany creates and injects callback proxies automatically based on annotations within the service implementation class. The implementation class uses the `@Callback` annotation to specify how injection should be performed, using either of the following styles:

- *A public or protected field*—The callback proxy will be injected into the field. Here's an example of declaring a field for callback injection:

```
@Callback
protected CreditCardSecurity ccSecurity;
```

- *A public or protected setter method*—The callback proxy will be injected by calling the setter method. The following code shows a setter method declared for callback injection:

```
@Callback
public void setCardSecurity(CreditCardSecurity ccSecurity) {
    ....
}
```

The following listing shows an example Java implementation for the credit card payment service that uses `@Callback` to annotate a field in the Java implementation.

Listing 5.3 The implementation for the bidirectional credit card payment service

```
public class CreditCardPaymentCallbackImpl implements CreditCardPayment {

    @Callback
    protected CreditCardSecurity ccSecurity;

    public String authorize(CreditCardDetailsType card,
                            float amount) {
        if (amount > 1000) {
            for (int i = 0; i < 3; i++) {
              String pwd = ccSecurity.
              ➥ checkSecurity("Enter password");

              if (verifyPassword(card, pwd)) {
                  break;
              }
              if (i == 2) {
                  return "BadPassword";
              }
            }
        }
        makePayment(card, amount);
        return "OK";
    }

    private boolean verifyPassword(
                        CreditCardDetailsType card,
                        String pw) {
        ....
    }

    private void makePayment(
                      CreditCardDetailsType card,
                      float amount) {
        ....
    }
}
```

Injected with a callback proxy

Multiple callbacks can be made

Details not shown

Details not shown

For implementations with a single callback service, Tuscany injects callback proxies into all fields and setter methods, marked with `@Callback`, that match the type of the callback interface for the bidirectional service. The names of fields or setter methods marked

with `@Callback` aren't important. In `CreditCardPaymentCallbackImpl` in listing 5.3, the callback interface is `CreditCardSecurity` and there's one matching callback field, `ccSecurity`. For every call to the `authorize()` method, Tuscany creates a new instance of `CreditCardPayment` and injects the `ccSecurity` field with a callback proxy for the client making the call. `CreditCardPaymentCallbackImpl` can use the proxy to make one or more callbacks to the client, or it may not call back to the client at all.

If the implementation has no callback fields or setter methods that match its callback interface, no callback proxies are injected. This can happen if the implementation has no need to make callbacks, even though the bidirectional interface permits it.

If a service implementation class offers more than one bidirectional service, a call to one of these services will cause callback proxies to be injected only into the callback fields and setter methods that match the type of the callback interface for the current service call. It's also possible for a service implementation class to offer both a bidirectional service and a nonbidirectional service, in which case callback proxies are injected only when the bidirectional service is called. In these situations, when different proxies can be injected by different calls to the service, it's important to not call a callback proxy without first checking that the proxy has been injected, that is, that it's not null. In our payment sample the proxy is not checked for null because the `CreditCardPaymentCallbackImpl` class offers a single bidirectional service, so the callback proxy will always be injected.

Next we'll look at how a callback client for a bidirectional service is implemented in the Java language.

5.8.4 The client programming model for callbacks

Listing 5.4 shows an example Java implementation for the Payment component. It's a callback client for the CreditCardPayment component shown in listing 5.5. The code in listing 5.4 is based on listing 5.1 with changes to accommodate callbacks from the bidirectional service.

Listing 5.4 The client implementation for the bidirectional credit card payment service

```
public class PaymentCallbackImpl implements Payment,
                                  CreditCardSecurity {      <-- Callback interface added

    ....

    public String makePaymentMember(String customerId,
                                    float amount) {
        ....
    }

    public String checkSecurity(String securityPrompt) {   <-- Callback method
        ....
    }
}
```

The code in listing 5.4 has two additions to the code in listing 5.1 because CreditCardPayment now has a bidirectional service: `PaymentCallbackImpl` implements an additional interface, `CreditCardSecurity`, and it contains an implementation of the method `checkSecurity()` defined by that interface. These additions enable `PaymentCallbackImpl` to handle callback invocations made by a bidirectional service of type CreditCardPayment. The majority of the code from listing 5.1 has not been repeated for clarity.

You've seen how to use the basic programming model for bidirectional interfaces and callbacks in Java implementations. Now it's time to look at some other features that you can use in Java implementations to customize various aspects of a callback interaction.

5.8.5 Getting the callback proxy from the request context

In listing 5.3, the callback proxy was injected into the service implementation instance. This works if a new service implementation instance is created for each request to the service, because each instance will contain injected callback proxies targeted at the service whose request created the instance. To put this another way, there will be a one-to-one relationship between service requests, implementation instances, and injected proxies.

Other situations don't have this simple one-to-one relationship. For example, a composite-scoped implementation has a single instance that's used to process all service requests for the component. Callback proxies are injected into the component instance when it's first created and are not reinjected, and so these proxies will be correct only for the first client that calls the component.

For composite-scoped components, multiple service requests can even run concurrently on different threads within the same component instance. If these requests are from different clients of a bidirectional service, the callback proxies for the requests will contain different destination addresses corresponding to the clients that made the service calls. If callback proxies for all incoming requests were injected into the same instance of a composite-scoped component, the results would be unpredictable because new incoming requests would overwrite previously injected callback proxies that might still be needed to make callbacks. The interaction diagram in figure 5.6 illustrates this problem.

In figure 5.6 there's only one instance of CreditCardPayment, so step ❹ would overwrite the callback proxy for Jim that was injected in step ❷. When request ❺ is made to check Jim's security password, it would be directed to John instead of Jim.

To avoid this problem, Tuscany doesn't overwrite previously injected callback proxies when new requests are dispatched to an existing implementation instance. But this still causes problems for multiclient scenarios. If injected proxies retain their original values instead of being overwritten, the callback to Jim would go to the correct destination, but callbacks intended for John would go to Jim instead.

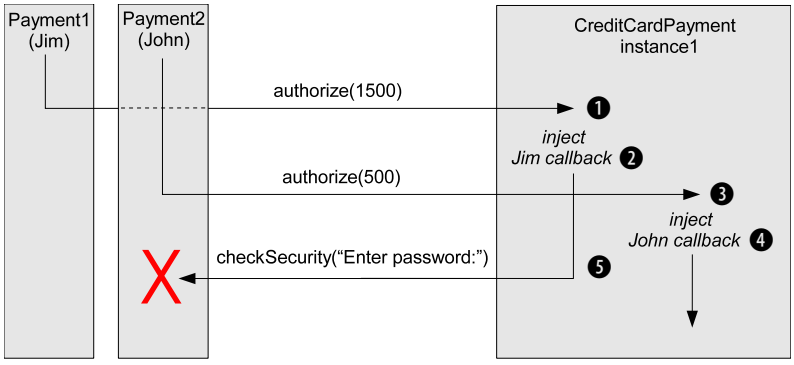


Figure 5.6 Two different client components are processing payments for Jim and John using the same CreditCardPayment component. For each payment authorization request, the same instance of the CreditCardPayment component is used. The injected callback proxies overwrite each other in the CreditCardPayment instance, and the callback is made to the wrong client component.

The solution is to write the implementation so that it doesn't use proxy injection in these cases. Instead, the implementation obtains a callback proxy from the current request context using the `RequestContext` Java SCA API, as follows:

```
@Context
protected RequestContext rqContext;

public String authorize(CreditCardDetailsType card, float amount) {
    ....
    CreditCardSecurity ccSecurity = rqContext.getCallback();
    String pwd = ccSecurity.checkSecurity("Enter password");
    ....
}
```

The `@Context` annotation can be placed on a field or setter method of type `RequestContext`. In this example we've placed it on the `rqContext` field. Tuscany will inject this field with a request context object which has access to the current service request. The implementation uses the `getCallback()` method on the request context object to get the callback proxy for the current request.

You may be thinking that we've just moved the problem because we're now injecting a request context object when the component instance is first created. But the request context code is thread safe and ensures that all callbacks go to the correct client, no matter how many clients are making concurrent calls to the same service.

Next we'll look at how a bidirectional service can delegate or defer callbacks using `CallableReference` objects.

5.8.6 Using callable references to provide callback flexibility

So far we've looked at callbacks made by a bidirectional service implementation before the original service request returns. It's also possible for the service implementation to

delegate the callback responsibility to some other service or for the callback to be made asynchronously sometime after the original service request has returned. To support these cases, Tuscany supports SCA `CallableReference` objects in Java implementations.

A `CallableReference` is an object that represents a callback. It provides methods to retrieve various kinds of information about the callback. `CallableReference` objects can also be passed on service method calls or stored persistently for later retrieval. A `CallableReference` is serializable, in the Java language sense, and so any mechanisms you use to store serialized Java objects can store a `CallableReference`.

To illustrate these scenarios, we'll use a different form of the CreditCardPayment service, called CreditCardPaymentConfirm. This has a bidirectional interface with a different callback interface from the previous example. The following listing shows that instead of using the callback interface to prompt for a security password, we'll use the callback interface to confirm successful payment and provide a payment reference.

Listing 5.5 Using a `CallableReference` object to delegate the callback

```
@Remotable
public interface CreditCardConfirm {
    void confirmPayment(String paymentRef);
}

@Remotable
@Callback(CreditCardConfirm.class)
public interface CreditCardPaymentConfirm {
    String authorize(CreditCardDetailsType ccDetails, float amount);
}

public class CreditCardPaymentConfirmImpl implements
                                                CreditCardPaymentConfirm {

    @Callback
    protected CallableReference<CreditCardConfirm> callbackRef;

    @Reference
    protected CardCompany cardCompany;

    public String authorize(CreditCardDetailsType card,
                            float amount) {
        cardCompany.makePayment(card, amount, callbackRef);
        return "InProgress";
    }
}
```

In listing 5.5, `CreditCardPaymentConfirmImpl` has a reference, `cardCompany`, for a service of type `CardCompany` that will make the real payment. The `@Callback` annotation is placed on a field of type `CallableReference<CreditCardConfirm>`, which causes Tuscany to inject a `CallableReference` object instead of a callback proxy. `CreditCardPaymentConfirmImpl` calls the CardCompany service to make the payment and passes the `CallableReference` object on this service call. The CardCompany service makes the payment, generates a payment reference, and makes a callback confirmation to the

original client using the `CallableReference` object that it was passed. The code to make the callback would look like the following:

```
callbackRef.getService().confirmPayment(paymentRef);
```

This code uses the `getService()` method of the `CallableReference` object to obtain a callback proxy that's used to call the client's `confirmPayment()` method.

You've seen how a bidirectional service can use `CallableReference` objects to provide more flexibility for how and when callbacks are made. Now we'll look at how a client can identify and correlate a specific callback using its callback ID.

5.8.7 Using a callback ID to identify a specific callback

In some cases the client may need to correlate a specific service request with a callback made by the service as a result of that request. SCA provides this capability by generating a unique callback ID for every call to a bidirectional service and making this callback ID available to callbacks that a service makes on behalf of the original call.

By default, the callback ID for a call to a bidirectional service is generated by Tuscany and can be obtained by the client method making the call, the service implementation method called by the client, and the client method that receives the callback from the service. Alternatively, the client method can choose to set its own unique value for the callback ID. This value would usually be something that's meaningful in business terms, such as an order number or payment reference. If the client implementation wants to customize the callback ID, it can do this by obtaining a `ServiceReference` object for the bidirectional service and calling the `setCallbackID()` method on this `ServiceReference` object. See section 5.8.8 for more information about `ServiceReference` objects.

The next listing is a modification of the `PaymentCallbackIDImpl` class showing how the client implementation can set and retrieve a callback ID.

Listing 5.6 Client implementation that sets and retrieves the callback ID

```
public class PaymentCallbackIDImpl implements Payment,
                              CreditCardSecurity {
    ....

    @Reference(name="creditCardPayment")
    protected ServiceReference<CreditCardPayment> cpRef;

    @Context
    protected RequestContext rqContext;

    public String makePaymentMember(String customerId,
                                    float amount) {
        ....
        String auditID = "987654321"
        cpRef.setCallbackID(auditID);
        String status = cpRef.getService().authorize(
                            ccDetails, amount);
```

1 Reference injected as ServiceReference

Audit ID created

2 Callback ID set here

```
        ....
    }

    public String checkSecurity(String securityPrompt) {
        Object cbID = rqContext.getServiceReference()
                               .getCallbackID();
        AuditRecord rec = getPaymentAuditRecord(cbID);
        ....
    }
}
```

3 Callback ID retrieved here

Callback ID holds audit ID

Listing 5.6 shows how the callback ID can be used in a callback to correlate the callback with the original service request and retrieve relevant information for processing the callback. Some of the detail has been omitted in order to focus on the code that deals with the callback ID.

First we'll need a `ServiceReference` object in order to set the callback ID. The easiest way to obtain a `ServiceReference` object is to declare the injected callback as being of type `ServiceReference` 1.

To customize the callback ID that will be used for the `authorize()` call and its associated callback, `checkSecurity()`, we'll call the `setCallbackID()` method 2 on the `ServiceReference` object, passing the dummy `audit ID` for this payment.

The `checkSecurity()` callback method retrieves the callback ID by calling `getServiceReference()` on the `RequestContext` object and calling `getCallbackID()` on the returned `ServiceReference` object 3. The callback ID is the audit ID for this payment, and we'll use this to retrieve the audit record holding the saved payment details.

We've looked at various ways that the service and client can customize the callback interaction. In the next section, you'll see how the client can redirect callbacks so that they go to another service.

5.8.8 Redirecting the callback to another service

A client implementation that calls a bidirectional service can choose not to handle callbacks itself, but redirect callbacks to another service instead. To do this, the client implementation obtains a `ServiceReference` object for the bidirectional service and calls the `setCallback()` method on this `ServiceReference` object, passing another `ServiceReference` object identifying a service that will handle callbacks from the bidirectional service. The following code is a modification of the code in listing 5.6, showing an example of this.

Listing 5.7 Delegating callbacks to another service

```
public class PaymentCallbackRedirectImpl implements Payment {
    ....

    @Reference(name="creditCardPayment")
    protected ServiceReference<CreditCardPayment> cpRef;

    @Reference(name="creditCardSecurity")
    protected ServiceReference<CreditCardSecurity> csRef;
```

Service that handles callbacks

```
    public String makePaymentMember(String customerId,
                                    float amount) {
        ....
        cpRef.setCallback(csRef);                          <-- Redirect callbacks
        String status = cpRef.getService().authorize(          to another service
                            ccDetails, amount);
        ....
    }
}
```

In listing 5.7, `PaymentCallbackRedirectImpl` doesn't implement the `CreditCardSecurity` interface or the `checkSecurity` method. Instead, it declares a `ServiceReference` object, `cpRef`, for the credit card payment service and another `ServiceReference` object, `csRef`, for a service that will handle `CreditCardSecurity` callbacks instead of `PaymentCallbackRedirectImpl`. Before calling the credit card payment service, `PaymentCallbackRedirectImpl` calls the `setCallback()` method on `cpRef` so that future calls through this reference will have callbacks redirected to the service identified by `csRef`. When the credit card payment service makes callbacks to get security passwords, it calls the service identified by `csRef` directly, and `PaymentCallbackRedirectImpl` won't be involved in any way.

In this section you've seen how to declare bidirectional interfaces and implement services and clients that use callbacks, and you've looked at the SCA APIs that Java implementations can use to customize callback interactions. Next we'll look at how Java implementations handle conversations.

5.9 Holding conversations

You saw in chapter 4 that SCA and Tuscany support the conversational service interaction pattern. In SCA, holding a conversation with a service means that all of the messages sent to a service from a specific reference arrive at the same service instance. This allows the service to maintain context as it processes a sequence of messages. Components implemented with Java are provided with a number of features for creating and controlling conversations. If you don't need to know the details of how to create and control conversations with Java components, you can safely skip forward to section 5.10.

5.9.1 Defining and controlling conversations in Java implementations

Table 5.5 shows the implementation.java-specific features that allow you to define and control a conversational service.

Table 5.5 The SCA features used to control conversational behavior

Feature	Description
`@Conversational`	Marks a service or service interface as being conversational.
`@EndsConversation`	Marks a service operation as being the last operation in the conversation.

Table 5.5 The SCA features used to control conversational behavior *(continued)*

Feature	Description
`@ConversationID`	Identifies a field to be injected with the conversation ID of the current conversation.
`@Scope("CONVERSATION")`	Sets the scope of a service implementation to be `CONVERSATION`. A new component instance will be created at the start of each new conversation, and the instance will remain until the conversation ends. All messages that are part of the conversation, as determined by the conversation ID, will be directed to the same service instance.
`CallableReference.isConversational()`	Returns `true` if the reference points to a conversational service.
`CallableReference.getConversation()`	Retrieves the conversation object, which implements the `Conversation` interface for getting the conversation ID and ending the conversation manually.
`Conversation.getConversationID()`	Returns the conversation ID for the conversation represented by this conversation object.
`Conversation.end()`	Ends the conversation.
`ServiceReference.getConversationID()`	Retrieves the conversation ID for the current conversation for this service reference.
`ServiceReference.setConversationID()`	Sets the conversation ID for the conversation about to start on this service reference.
`@ConversationAttributes( maxAge="10 minutes", maxIdleTime="5 minutes", singlePrincipal=false)`	Associates conversation configuration with the service implementation.

You'll notice that some of these features rely on Java annotations, and some of them are supported using a Java API. Let's look at the annotations first.

5.9.2 Starting, using, and stopping conversations using annotations

Chapter 4 gave an overview of the conversational interaction pattern, and here we'll look at the Java language–specific features. In keeping with chapter 4, we'll move away from the payment example briefly and review the CartStore component from the shoppingcart contribution. As a reminder, the following listing shows how a service interface is marked as being conversational.

Listing 5.8 The conversational `CartStore` interface

```
@Conversational
public interface CartStore{
    void addTrip(TripItem trip);
    void removeTrip(TripItem trip);
    TripItem[] getTrips();
```

Marks interface as conversational

Operations within a conversation

```
    @EndsConversation
    void reset();
}
```

Conversation ends after this operation

The `@Conversational` annotation in listing 5.8 identifies this interface as being conversational. The first call to any of its operations starts a new conversation. Subsequent calls to this interface's operations continue the conversation until the operation marked with the `@EndsConversation` annotation is called. At this point the conversation ends.

The next listing shows the service implementation itself. Note that the scope of the component is set to `CONVERSATION` so a new instance is created for each conversation.

Listing 5.9 The conversational `CartStore` component implementation

```
@Scope("CONVERSATION")
@ConversationAttributes(maxAge="10 minutes",
                        maxIdleTime="5 minutes",
                        singlePrincipal=false)
@Service(interfaces={CartStore.class})
public class CartStoreImpl implements CartStore{

    @ConversationID
    protected String cartId;

    private List<TripItem> trips = null;

    @Init
    public void initCart() {
        trips = new ArrayList<TripItem>();
    }

    @Destroy
    public void destroyCart() {
        trips = null;
    }

    public void addTrip(TripItem trip) {
        trips.add(trip);
    }

    public void removeTrip(TripItem trip) {
        trips.remove(trip);
    }

    public TripItem[] getTrips(){
        return trips.toArray(new TripItem[trips.size()]);
    }

    public void reset(){
        trips.clear();
    }
}
```

each co

1 Control conversation behavior

2 Conversation ID injected automatically

3 Operation called at conversation start

Initialize conversational instance

4 Operation called at conversation end

Destroy conversational instance

5 Causes conversation to end

The conversation starts when the first message arrives. A component instance is created and the conversation ID is injected into the field marked with the `@ConversationID` annotation, `cartId` ❷. The operations annotated with `@Init` ❸ are also called. This is an opportunity to arrange for conversational state to be stored.

Now that the conversational component has been created, any of its operations that are called through the reference that sent the first message will arrive at the same component instance. The conversational state is therefore easily available.

When the operation marked with `@EndsConversation` in the interface, `reset` in this case ❺, is called, the conversation ends. Any operations marked with the `@Destroy` annotation ❹ will be called. This gives you an opportunity to free up any resources associated with the conversation. Finally, the component instance is removed.

You don't have to use `@Init` and `@Destroy` with conversational services, but it's a convenient way to add code that runs automatically at the start and end of a conversation.

The lifetime of a conversation can also be controlled by conversational attributes ❶. The `maxAge` attribute determines how long the conversation can live for before it's stopped automatically. The `maxIdleTime` attribute determines how long the conversation can live between incoming messages before it's stopped automatically. The `singlePrincipal` attribute should allow you to indicate that a conversation can be used only by the client that started it, but this attribute isn't supported by Tuscany at the moment.

Some extra features of conversations can be controlled via the SCA Java API.

5.9.3 Controlling conversations using the SCA Java API

As an alternative to relying on injection to access the conversation ID, you can set and retrieve it using the `ServiceReference` Java API. For example, here's code that retrieves a reference to the CartStore component and starts a new conversation with it:

```
ServiceReference<CartStore> cartStore =
        componentContext.getServiceReference(CartStore.class,
                                             "cartStore");
cartStore.setConversationID(UUID.randomUUID().toString());
cartStore.addTrip(bookedTrip);
```

Once you have a service reference to a conversational service, you can continue to call operations on it or even pass it to other services so that they can call operations as part of the conversation. The conversation continues until it's ended by, in this case, calling the `reset` operation.

A conversation object can be retrieved from a service reference. This object implements the `Conversation` interface defined by the SCA Java API, which provides another way of retrieving the conversation ID and also provides an `end` operation that allows a conversation to be terminated prematurely.

We've finished looking at how to use the conversation pattern with components implemented in the Java language, so let's move on and look at how service references can be obtained and passed from one component to another.

5.10 Passing SCA service references

It's often desirable to get a reference to a service instead of using the proxy injected automatically into a component reference. This is particularly important when collaborating components need to exchange service references. We'll first look at a scenario where this can be useful and then look at how to retrieve, pass, and use a service reference.

5.10.1 A service reference–passing scenario

Let's introduce a variant of the Payment component's interaction with the EmailGateway and CreditCardPayment components. You can find these in the `payment-java-reference-pass` sample contribution. Figure 5.7 shows the various interactions. Let's assume that the authorize operation of the CreditCardPayment is time consuming and that we don't want to prevent the Payment component from continuing with its processing. We do, though, want to ensure that whenever the credit card transaction is done, we send an email notification.

The Payment component must get a service reference that refers to the EmailGateway service and pass this to the CreditCardPayment component when the authorize operation is called, and then the CreditCardPayment component uses this reference to call the EmailGateway component when processing is complete.

Let's find out how the Payment component can retrieve a service reference to the EmailGateway component.

5.10.2 Retrieving service references

The first way to retrieve a service reference is to declare an SCA reference with the Java type `ServiceReference`. For example, in the following snippet, a `ServiceReference` for the EmailGateway component service will be injected into the emailGateway field

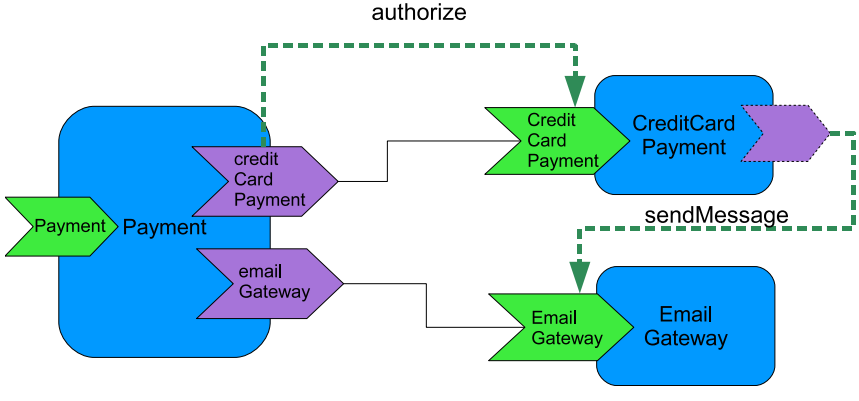


Figure 5.7 By using service references passed between SCA components, the CreditCardPayment component can be asked to call the EmailGateway component.

by the runtime. Please note that the parameterized type for the `ServiceReference` interface is the business interface EmailGateway.

```
@Reference
protected ServiceReference<EmailGateway> emailGateway;
```

The second way to retrieve a service reference is through the ComponentContext SCA Java API. To do this, the component context must be injected into the Payment component.

```
@Context
Protected ComponentContext context;
...
ServiceReference<EmailGateway> emailGateway =
     context.getServiceReference(EmailGateway.class, "emailGateway");
```

As you can see, the component context is then used to retrieve the service reference by name. Now we'll need to pass the service reference to the CreditCardPayment component.

5.10.3 Passing a service reference to another component

The service reference for emailGateway can be passed to the CreditCardPayment component explicitly as a parameter on the `authorize` method in the following way:

```
void authorize(CreditCardDetailsType cc,
               float amount,
               ServiceReference<EmailGateway> emailGateway,
               String emailAddress);
```

The code in the Payment component to call the `authorize` method passing a service reference would be as follows:

```
@Reference
protected ServiceReference<EmailGateway> emailGateway;

@Reference
protected CreditCardPayment creditCardPayment;

public String makePaymentMember(String customerId, float amount) {
    creditCardPayment.authorize(ccDetails,
                                amount,
                                emailGateway,
                                emailAddress);
    ...
}
```

This gives the CreditCardPayment component the information it needs to call the EmailGateway component when the authorization is complete. Let's see how the CreditCardPayment component uses the service reference it receives.

5.10.4 Making a call via a service reference

The Payment component has passed in a ServiceReference for the EmailGateway through the parameters of the `authorize` method. Once credit card processing is

complete, the CreditCardPayment component calls the emailGateway to send out the notification email as follows:

```
emailGateway.getService().sendEmail(…);
```

The EmailGateway service now sends out an email to notify the customer of the result of the transaction, and the Payment component doesn't have to wait for this to happen.

We've talked a lot about how to build a component implementation in Java when everything is working as expected. We haven't yet covered what to do when things go wrong.

5.11 Handling errors

When an SCA component interacts with other services, error conditions may occur. In general, there are two types of exceptions: business exceptions and SCA runtime exceptions. We'll cover both how to define and handle business exceptions and how to handle SCA runtime exceptions, so let's start by looking at business exceptions.

5.11.1 Business exceptions

Business exceptions are defined as checked exceptions on the Java interface or faults on the WSDL port type and are thrown by the component implementation. Business exceptions are part of the contract for an SCA service or reference and are used to represent and carry data for an error condition that occurs in the business logic. For example, you can define a `CustomerNotFoundException` in the Java language to indicate that a customer ID can't be found in the customer registry or an `authorizationFailure` fault in WSDL to indicate that a credit card transaction can't be authorized because the card number is invalid.

Let's look at two examples to explain how business exceptions can be declared on the business interface and how to handle them in the component implementation class.

The first example uses a local Java interface. Component implementations written in the Java language use the same error-handling techniques that an ordinary Java program uses. For example, the customer registry component locates customer details based on a customer identifier. If an invalid customer identity is passed in, then the customer registry component must be able to report that the requested customer can't be found. To indicate this error state, we'll create a `CustomerNotFoundException` type, which contains the `customerId` in question.

```
public class CustomerNotFoundException extends Exception {
    private String customerId;
    ...
}
```

We can then extend the service interface to indicate that this service may return this exception under certain conditions. Because we're working with implementation.java, and the service interface is described in the Java language, this is a matter of adding the exception to appropriate operations in the Java `CustomerRegistry` interface.

```
public interface CustomerRegistry {
    Customer getCustomer(String id) throws CustomerNotFoundException;
}
```

The `CustomerRegistryImpl` class throws the `CustomerNotFoundException` if the lookup of the customer registry doesn't return a record for a given `customerId`.

A client of this service should test for this error condition. Again, because we're working in the Java language, we can use the normal Java language techniques to do this. Note that in this case we also check for any unchecked exceptions that might be received.

```
public String makePaymentMember(String customerId, float amount) {
    Customer customer = null;

    try {
        customer = customerRegistry.getCustomer(customerId);
    } catch (CustomerNotFoundException ex) {
        return "Payment failed due to " + ex.getMessage();
    } catch (Throwable t) {
        return "Payment failed due to system error " + t.getMessage();
    }
}
```

When dealing with business exceptions, the remotable interface is more complicated than the local interface. For remote service interactions, the business exceptions are sent back to the client side within protocol messages. Different bindings may have different ways to pass exception information. Let's extend the CreditCardPayment component to illustrate remote error handling following the JAX-WS rules for the Web Services binding.

We'll first define the faults that the CreditCardPayment service interface returns by updating CreditCardPayment.wsdl. Listing 5.10 shows the related definitions.

Listing 5.10 Adding fault types to the CreditCardPayment WSDL interface description

```
<wsdl:definitions …>
  <wsdl:types>
    <xsd:schema …>
      ...
       <xsd:element name="authorizeFault">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="errorCode" type="xsd:string"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
     </xsd:schema>
   </wsdl:types>
   ...
   <wsdl:message name="AuthorizeFault">
     <wsdl:part name="parameters"
                element="tns:authorizeFault"/>
   </wsdl:message>

   <wsdl:portType name="CreditCardPayment">
     <wsdl:operation name="authorize">
       <wsdl:input message="tns:AuthorizeRequest"/>
```

1 Global fault element

2 Fault message

```
        <wsdl:output message="tns:AuthorizeResponse"/>
        <wsdl:fault name="authorizationFailure"
                    message="tns:AuthorizeFault"/>
      </wsdl:operation>
    </wsdl:portType>
  <wsdl:binding name="CreditCardPaymentBinding"
                type="tns:CreditCardPayment">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="authorize">
          <soap:operation soapAction=
                "http://www.tuscanyscatours.com/CreditCardPayment/authorize"/
      >
          <wsdl:input>
              <soap:body use="literal"/>
          </wsdl:input>
          <wsdl:output>
              <soap:body use="literal"/>
          </wsdl:output>
          <wsdl:fault name="authorizationFailure">
            <soap:fault name="authorizationFailure" use="literal"/>
          </wsdl:fault>
      </wsdl:operation>
  </wsdl:binding>
  ...
</wsdl:definitions>
```

3 Operation's fault message

4 Operation's fault message binding

In the WSDL we define a global element `authorizeFault` 1. A WSDL message is added with a part that points to the `authorizeFault` element 2. Finally, a WSDL fault is added under both the portType/operation 3 and the binding/operation 4.

The JAX-WS wsimport tool is used to generate Java interfaces from WSDL. Chapter 9 discusses how to use this tool in more detail. When wsimport is run against this WSDL, three related artifacts are generated:

- A JAXB class `payment.creditcard.AuthorizeFault` for the fault element (this is no different than a regular parameter or return type).
- A Java exception `payment.creditcard.AuthorizeFault_Exception` that wraps the fault. See listing 5.11.
- A Java interface `payment.creditcard.CreditCardPayment` with the `authorize()` method that throws the `payment.creditcard.AuthorizeFault_Exception`.

Listing 5.11 The generated `payment.creditcard.AuthorizeFault_Exception`

```
package payment.creditcard;

import javax.xml.ws.WebFault;


@WebFault(name = "authorizeFault",
    targetNamespace = "http://www.tuscanyscatours.com/CreditCardPayment/")
public class AuthorizeFault_Exception extends Exception
```

```
{
    private AuthorizeFault faultInfo;
    public AuthorizeFault_Exception(String message,
                                    AuthorizeFault faultInfo) {
        super(message);
        this.faultInfo = faultInfo;
    }

    public AuthorizeFault_Exception(String message,
                                    AuthorizeFault faultInfo,
                                    Throwable cause) {
        super(message, cause);
        this.faultInfo = faultInfo;
    }

    public AuthorizeFault getFaultInfo() {
        return faultInfo;
    }

}
```

The `@WebFault` annotation shown above the class definition in listing 5.11 is a JAX-WS annotation added to indicate that the Java exception represents a WSDL fault.

The `authorize()` method of the `CreditCardPayment` interface now has a `throws` clause of `AuthorizeFault_Exception` to indicate that in some cases this method can throw this exception:

```
public String authorize(@WebParam(name ="CreditCard", targetNamespace ="")
                        CreditCardDetailsType creditCard,
                        @WebParam(name = "Amount", targetNamespace = "")
                        float amount)
    throws AuthorizeFault_Exception;
```

The component implementation class, `CreditCardPaymentImpl`, now has the code to create and throw the `AuthorizeFault_Exception`, as follows:

```
public String authorize(CreditCardDetailsType creditCard,
                        float amount)
    throws AuthorizeFault_Exception {

    if (creditCard != null) {
        ...
    } else {
        ObjectFactory factory = new ObjectFactory();
        AuthorizeFault fault = factory.createAuthorizeFault();
        fault.setErrorCode("001 - Invalid card");
        AuthorizeFault_Exception ex =
            new AuthorizeFault_Exception("Invalid card",
                                         fault);
        throw ex;
    }

    return "OK";
}
```

On the client side, where the credit card payment is called, the PaymentImpl component implementation captures the exception and extracts the error code:

```
try {
    status = creditCardPayment.authorize(ccDetails,
                                         amount);
} catch (AuthorizeFault_Exception e) {
    status = e.getFaultInfo().getErrorCode();
}
```

As you can see, even though the exception is now described in the WSDL interface and could pass across a remote binding, such as the Web Services binding, the Java implementation still deals with the Java exception types that it's familiar with. Regardless of which binding you use, the Tuscany runtime will ensure that the behavior remains consistent when errors arise.

5.11.2 SCA runtime exceptions

Unlike business exceptions, some error conditions aren't expected by the business logic and instead are caused by the Tuscany runtime, for example:

- There are configuration errors in the application's composite files.
- A target service isn't running or it can't be reached because of networking problems.
- Quality of service constraints are violated; for example, the user can't be authenticated or can't be authorized.

In most cases, SCA runtime exceptions can't be handled properly by the application code. Instead they're handled using strategies such as these:

- Don't catch the runtime exceptions and let them pop up the calling stack.
- Catch it and throw a business exception to indicate the abnormal system status.
- Catch it and resend the message or initiate a compensating action.

Of these, the best practice is to catch the exception and either throw it as a business exception described explicitly on your service interface or initiate a compensating action if you've designed your components to work in this way. Allowing system exceptions to work their way back through the stack of calling component references and services can be problematic. There's a good chance that system details from deep within the call stack will be exposed in unexpected places, with the obvious implications for security.

You've now had an extensive tour of the various SCA features you can exploit to configure components implemented using Java classes. Let's wrap up now before moving on to the look at other implementation types in the next chapter.

5.12 Summary

In the initial chapters of the book, you learned how SCA offers a technology-agnostic component model. Its support of various component implementation types enables different technologies to work with one another in an SCA composite application.

This chapter offered a technical perspective of the current Java component implementation type. We covered quite a lot of ground looking at the details of how you define services, references, and properties in a Java implementation. We also looked at how the callback and conversation interaction patterns described in chapter 4 are supported in Java implementations and how to handle errors. The subject of how to apply policy intents to Java component implementations using Java annotations will be covered in chapter 10, along with the wider discussion of policy support in Tuscany.

The main lesson of this chapter is that it's easy to do simple things using Tuscany and vanilla Java component implementations. Exploiting the more complex SCA features is also easy using SCA annotations. The Java language has been a particular focus of the SCA specifications and has the most features defined. Some, but not all, of these features are available in other component implementation types. Let's move on now and look at some of those other implementation types.

6 Implementing components using other technologies

This chapter covers

- Using Spring to implement components
- Using BPEL processes to implement components
- Using scripts to implement components

An SCA application is an assembly of components. Each component has an implementation type. In chapter 5 we looked at how you can implement components using the Java language implementation type (implementation.java). In this chapter we'll look at three more popular implementation types:

- implementation.spring
- implementation.bpel
- implementation.script

Both implementation.bpel and implementation.spring are described in SCA specifications. But implementation.script was created by the Tuscany community and appears in the Tuscany namespace. You can safely skip the implementation types that aren't of immediate interest to you.

The list of implementation types supported by the Tuscany runtime is growing as the community adds new extensions. Please check the project for the latest supported implementation types. Chapter 14 explains how a new implementation type can be added if the implementation type that you require isn't already available.

We'll continue using the Payment composite that we used in the last chapter to explain how Spring, BPEL, and scripting implementation types can be used. Let's start by looking at how you can use a Spring application context to implement an SCA component.

6.1 Implementing components using Spring

Applications developed with the Spring framework can participate in SCA composite applications by using the implementation.spring component implementation type. This is defined in the SCA Spring Component Implementation Specification (http://www.osoa.org/download/attachments/35/SCA_SpringComponentImplementationSpecification-V100.pdf?version=1). SCA components implemented using Spring application contexts can interact with other SCA components using any of the flexible Tuscany bindings. Feel free to skip this section if you're not planning to use Spring in your SCA application.

An SCA component implemented using implementation.spring is a Spring application that's created using a Spring application context. The Spring application context can contain one or more beans, and we'll show this in our example in this section. In Spring, a bean is the most basic configuration unit. In SCA, a component is the most basic configuration unit. An SCA composite provides a higher granularity of composition that can include SCA components implemented using Spring application contexts as well as SCA components written using other technologies such as scripting and BPEL.

The Spring application participates in an SCA composite by exposing its beans through WSDL or Java interfaces. Because Spring is a Java-based technology, service interfaces in the implementation are defined using Java interfaces. The mechanism by which these are mapped to component service and reference interfaces is the same as for implementation.java. Spring beans can participate in SCA composition as references or services without introducing any new metadata into the Spring application.

A Spring bean that provides a service implementation handles errors by throwing exceptions using normal Java language mechanisms. Assuming they're checked exceptions, the Java interface that describes the service will describe the exceptions that are thrown. In this way the Tuscany runtime can process exceptions and ensure that they're returned by binding extensions correctly.

We're going to use the Payment component again as the basis for the examples in this chapter. Before we delve into the details of how implementation.spring works, let's look at how the Spring-based Payment component fits into the overall TuscanySCATours composite application and the beans that the Spring application context could define. Figure 6.1 gives an overview of the payment part of the application found in the payment-spring sample contribution. Note that the CreditCard component can be found in the creditcard-payment-jaxb sample contribution.

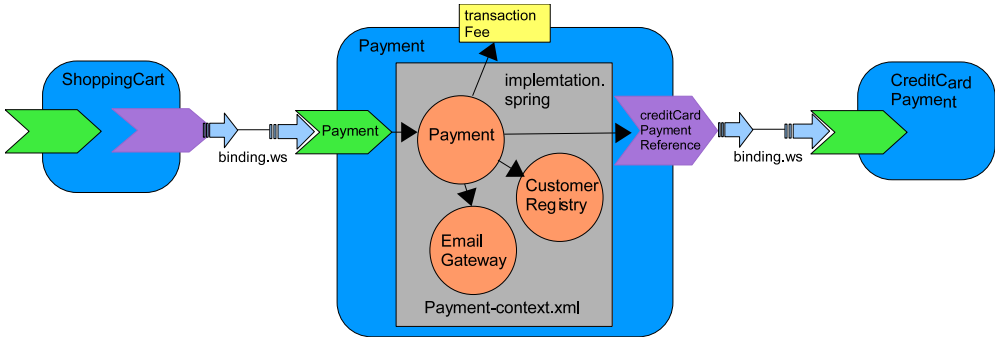


Figure 6.1 The Payment component implemented using implementation.spring, which, in turn, references a Spring application context, called Payment-context.xml, containing three beans

In this configuration the Payment component is implemented by three Spring beans, `Payment`, `CustomerRegistery`, and `EmailGateway`. The credit card payment function is provided by an external CreditCardPayment component accessed via an SCA reference. The Payment component uses the CreditCardPayment component to authorize the card payment and then uses the `EmailGateway` bean to send a payment confirmation email. The transaction fee property specifies how much TuscanySCATours charges to take a payment by credit card. With this configuration we're showing integration between local Spring beans and between Spring beans and SCA components.

An existing Spring application can be incorporated into an SCA composition using two different approaches. In the first approach, no changes are made to the Spring application context. The SCA runtime determines which beans are exposed as services and references. The second approach involves adding SCA metadata to the Spring application context to identify services and references explicitly. We'll first look at how to use an unmodified Spring context.

6.1.1 Using Spring services and references without SCA tags

The Payment-context.xml is a Spring application context defined in the sample contributions directory under contributions/payment-spring. It includes `Payment`, `CustomerRegistry`, and `EmailGateway` beans. The SCA runtime maps the Payment-context.xml application context to the SCA Payment component in the following way. Every top-level bean that has an ID or a name gets exposed as an SCA service. If a bean is nested within another bean, it's considered anonymous and isn't exposed as a service. Each Spring `<property>` that's of `ref` type gets exposed as an SCA reference. Each Spring `<property>` that's of `value` type gets mapped to an SCA component property. SCA services can be exposed without adding SCA-specific metadata to the Spring application context of the Payment component. Instead, the SCA composite file contains enough information to help the Tuscany runtime locate services and references in the payment application context. The sample that demonstrates how this is done can be found in the contributions directory of the travel-booking sample. Figure 6.2 provides a graphical view of how the `Payment` bean in the Payment-context.xml is mapped to SCA Payment component.

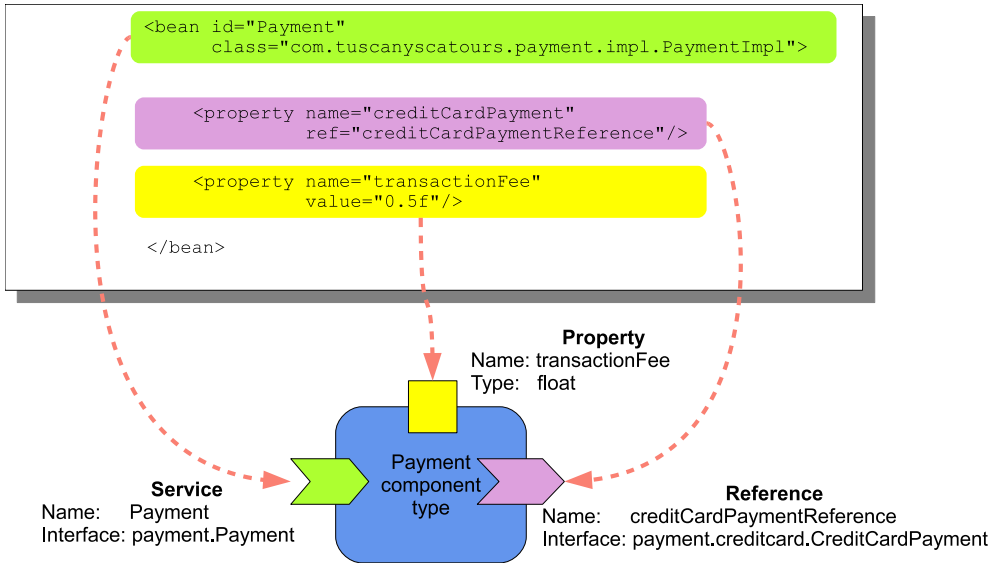


Figure 6.2 The mapping between the unannotated Spring Payment bean and SCA services, references, and properties

We've just looked at the Spring application context itself. In this case it doesn't have any SCA tags in it. Now let's look at the SCA composite description that references the Spring application context.

The payment.composite file from the payment-spring sample contribution defines the Payment component using the Payment-context.xml Spring application context for the component implementation.

```
<component name="Payment">
    <implementation.spring location="Payment-context.xml"/>
    <service name="Payment">
        <binding.ws uri="http://localhost:8080/Payment"/>
    </service>
    <reference name="creditCardPaymentReference">
        <binding.ws uri="http://localhost:8081/CreditCardPayment"/>
    </reference>
    <property name="transactionFee">1.23</property>
</component>
```

The Payment component is implemented using an implementation.spring type. The business logic for this component is identified by the `location` attribute, which, according to the SCA specifications, can point to either an archive file or a directory that contains the Spring application context file. Tuscany extends the specification and also allows the `location` attribute to be set to the name of a Spring application context file that resides in the same directory as the composite file.

The `<service>` element within the `<component>` element shows that Payment-context.xml provides a Payment service. Messages sent to this service are forward by

Tuscany to the Spring `Payment` bean. The `<reference>` element within the `<component>` element shows that `creditCardPaymentReference` is a web service and will be available as a reference within the Spring `Payment` bean.

It's important that the component service and reference names included in the composite file match the bean names that are specified in the Spring application context. Otherwise, the Tuscany runtime won't be able to match the SCA description of the component with the Spring application context.

Our first listing shows the Spring `Payment` application context for the SCA Payment component.

Listing 6.1 The Payment application context, Payment-context.xml, without SCA tags

```
<beans ...>
    <bean id="Payment"
          class="com.tuscanyscatours.payment.impl.PaymentImpl">
        <property name="creditCardPayment"
                  ref="creditCardPaymentReference"/>
        <property name="emailGateway" ref="EmailGatewayBean"/>
        <property name="transactionFee" value=".5f"/>
    </bean>

    <bean id="EmailGateway"
          class="com.tuscanyscatours.emailgateway.impl.EmailGatewayImpl">
    </bean>
</beans>
```

By default, all the beans defined in the Spring application context become available as services. Therefore, the `Payment` bean is automatically available as a service. The first property defined for the `Payment` bean is named `creditCardPayment` and refers to a `creditCardPayment` web service that isn't shown here. The reference `creditCard-PaymentReference` is also referred to by the SCA Payment component in the composite file. The reference name `creditCardPaymentReference` in the composite file matches the name of the `creditCardPaymentReference` reference in the Spring application context. This is how the Tuscany runtime realizes which of the beans act as services or are references to other services. The second property defined for the `Payment` bean refers directly to the local `EmailGateway` bean defined in the same Spring application context, and the Tuscany runtime respects the local Spring reference in this case and doesn't try to reconfigure it.

In this example, the payment application context is unmodified. The Tuscany runtime identifies references and services for the SCA Payment component by first finding the Spring application context that provides the implementation for the Payment component. It then introspects it to determine services and references for the Payment component.

This default mechanism may not be ideal at all times. For example, a Spring application developer may want to control precisely which of the beans in the application context are exposed as SCA services. This is possible through custom SCA Spring tags.

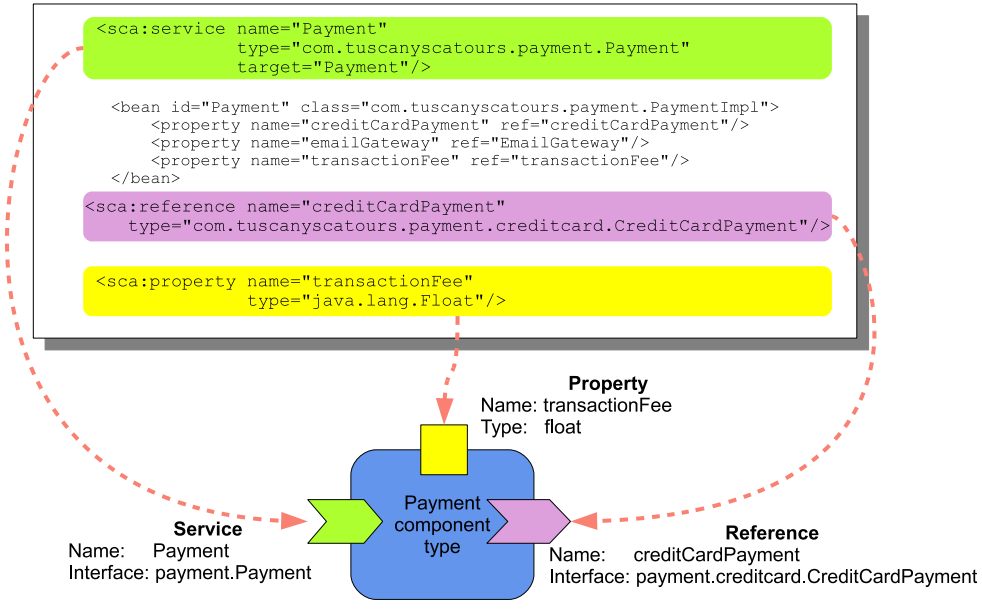


Figure 6.3 The mapping between the modified Spring application context and SCA services, references, and properties

6.1.2 Using Spring services and references with SCA tags

The Spring tags that SCA defines can be used in the application context to identify the beans that will be used as SCA services or references. The sample that demonstrates how this is done can be found in the contributions/payment-spring-scatag sample directory.

Figure 6.3 shows how Spring Payment-context.xml can be modified to include SCA extension tags that define SCA services, references, and properties.

As shown in the diagram, the `creditCardPayment` reference and Payment service are tagged with `sca:reference` and `sca:service`, respectively. These tags explicitly describe the artifacts from the Spring application context that can be used in the SCA composition. On the other hand, the `EmailGateway` bean isn't described using an SCA tag and can't be used in the composition. Notice that once SCA tags are used, the defaulting mechanism that was explained in the previous section does not apply. Again the local wiring to the `EmailGateway` bean is preserved. The next listing shows Payment-context.xml Spring with SCA tags for services, references, and properties.

Listing 6.2 The payment application context, Payment-context.xml, with SCA tags

```
<beans ...>
    <sca:service name="Payment"
                 type="com.tuscanyscatours.payment.Payment"
                 target="Payment"/>

    <bean id="Payment"
          class="com.tuscanyscatours.payment.impl.PaymentImpl">
```

```
        <property name="creditCardPayment" ref="creditCardPayment"/>
        <property name="emailGateway" ref="EmailGateway"/>
        <property name="transactionFee" ref="transactionFee"/>
    </bean>

    <bean id="EmailGateway"
          class="com.tuscanyscatours.emailgateway.impl.EmailGatewayImpl">
    </bean>

    <sca:reference name="creditCardPayment"
          type="com.tuscanyscatours.payment.creditcard.CreditCardPayment"/>

    <sca:property name="transactionFee"
                  type="java.lang.Float"/>
</beans>
```

We've looked at how services, references, and properties are mapped to a component's description in the composite file. Now let's look in more detail at how properties are set in the SCA composite.

6.1.3 Setting Spring properties

A Spring application can have configuration properties that control how beans should behave. You've seen in the previous payment.composite listing that the Payment component defines a property called `transactionFee`. Let's change the initial value of this property to `9.73`.

```
<component name="Payment">
     <implementation.spring location="Payment-context.xml"/>
     ...
     <property name="transactionFee">9.73</property>
</component>
```

The section of the Spring application context related to properties follows. Here we're using SCA tags in the Spring context file. Again, note that the names for the property, as it appears in the composite file and in the application context, need to match in order for Tuscany to map the Spring property to the SCA property.

```
<bean id="Payment" class="com.tuscanyscatours.payment.impl.PaymentImpl">
    <property name="creditCardPayment" ref="creditCardPayment"/>
    <property name="emailGateway" ref="EmailGateway"/>
    <property name="transactionFee" ref="transactionFee"/>
</bean>

...

<sca:property name="transactionFee"
              type="java.lang.Float"/>
```

If no SCA property tags are included in the Spring context, then the Tuscany runtime will try to match property elements in the SCA composite file with property names it finds in the Spring application context.

6.1.4 Using other SCA Java annotations

Tuscany has added support for annotations in SCA components that are implemented as Spring beans. This is an extension to the SCA Spring Component Implementation Specification. The idea behind this is to add SCA annotations to a Java implementation that will be used as a Spring bean. The annotations describe the behavior of the bean in the overall application composition. We talked about Java annotations in chapter 5. Currently, the basic SCA annotations `@Service`, `@Reference`, `@Property`, `@Init`, `@Destroy`, and `@ComponentName` are recognized in a Spring bean. Because Spring also defines an `@Service` annotation, you should fully qualify the SCA version of the annotation. For example, the following listing shows the `PaymentImpl` bean that's annotated with SCA annotations `@Service` and `@Reference` to identify services and references.

Listing 6.3 The `Payment` Spring bean with SCA annotations

```
@org.osoa.sca.annotations.Service(Payment.class)
public class PaymentImpl implements Payment {

    protected CreditCardPayment creditCardPayment;
    protected EmailGateway emailGateway;
    protected float processingCharge = 0;

    @Reference
    public void setCreditCardPayment(CreditCardPayment creditCardPayment) {
        this.creditCardPayment = creditCardPayment;
    }

    public void setEmailGateway(EmailGateway emailGateway) {
        this.emailGateway = emailGateway;
    }
}
```

Support for more annotations may be introduced in the future.

6.1.5 Finding the Spring application context

We mentioned earlier that the `location` attribute of the `<implementation.spring>` element identifies where the implementation for the implementation.spring type can be found. A Spring context that's intended to be used as a component implementation must be in the same contribution as the composite file containing the component being implemented. The Tuscany runtime uses the following rules to find it:

- If the `location` attribute points directly at a Spring context file, then this file is used as the component implementation.
- If the `location` attribute represents an archive file, the META-INF/MANIFEST.MF file is read from the archive.
- If the `location` attribute represents a directory, the file META-INF/MANIFEST.MF is expected to appear under this directory.

When a MANIFEST.MF file has been located, the following rules apply as defined by the SCA Spring Component Implementation Specification:

- If the manifest file contains a header of the following format, then the set of paths specified in the header identify the Spring context configuration files: Spring-Context ::= path(';' path)*. In this case, *path* is a relative path with respect to the location URI.
- If the MANIFEST.MF file isn't present or there is no Spring-Context header in the MANIFEST.MF, by default Tuscany will build the application context using the application-context.xml file available in the META-INF/spring directory.

Within a Spring application context, further location information can be specified. For example, the Spring framework's `ClassPathXmlApplicationContext` bean can be used to reference further Spring application contexts. When used in an application context that's being used to implement an SCA component in Tuscany, the paths to the application contexts that will be used are assumed to be given relative to the root of the current contribution.

The Spring `FileSystemXmlApplicationContext` bean isn't supported when implementing SCA components in Tuscany because this could allow the application context to reference other application contexts outside the SCA contribution.

Let's conclude this section with a reminder that because implementation.spring is defined by the SCA specifications and isn't a Tuscany extension, it resides under the OSOA SCA specification namespace (http://www.osoa.org/xmlns/sca/1.0). The implementation type that supports BPEL processes is also defined in an SCA specification, so we'll look at that next.

6.2 Implementing components using BPEL

The SCA Client and Implementation Model Specification for WS-BPEL describes the extension implementation type implementation.bpel and can be found at the following URL: http://www.osoa.org/download/attachments/35/SCA_ClientAndImplementationModelforBPEL_V100.pdf. The implementation.bpel type plays the same role as any other implementation extension in SCA, as shown in figure 6.4. Feel free to skip this section if you're not planning to use BPEL in your SCA application.

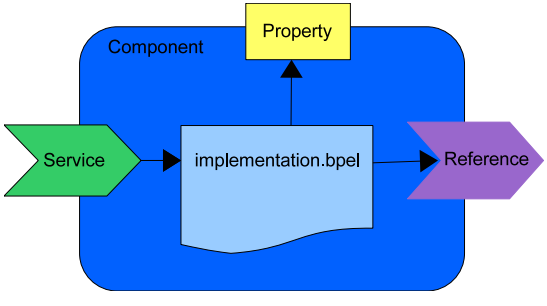


Figure 6.4 Implementation.bpel provides a WS-BPEL implementation for an SCA component.

Generally a BPEL process is used to describe business processes in software. Typically a business process captures a sequence of ordered activities that must be performed. Decisions made during the process are captured as branches in the sequence of activities.

The idea of developing business process management software to coordinate business-level activities grew out of a desire to describe and automatically manage business processes at a higher level than in a programming language such as Java.

An application based on SOA principles typically consists of a number of loosely coupled interacting services. The processes within a business define the way that these services interact, and hence BPEL and SOA fit well together. SCA describes the network of services, whereas BPEL can be used to coordinate their activities.

As you've seen in the previous chapter, the job of organizing a set of SCA service interactions, for the Java language programmer, is a simple matter of writing a component implementation in the Java language that calls the right operations on the available references in the correct order. But BPEL has been specifically designed to express a business process as a sequence of web service calls, so it offers a useful alternative.

In this section we'll take a quick look and refresh our memory of how BPEL processes are laid out in XML. This is important, because we'll then move on to look at how SCA and Tuscany map BPEL concepts to component services, references, and properties.

Don't worry if you're not a BPEL expert. This is the first time we've presented a component implementation technology that isn't based on the Java language. If you've reached this point because you're interested in alternative implementation technologies, then you'll be glad to know that the SCA bits remain the same. The part that changes is how these map onto the BPEL programming model. This is true for any component implementation technology. You just need to find the parts of the technology that represent services, references, and properties.

6.2.1 The structure of a BPEL process document

The Web Service Business Process Execution Language standards have been developed in the OASIS WSBPEL Technical Committee (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel). BPEL allows developers to describe a business process as a sequence of calls to web services. As a language for describing business processes, it has many things that a programmer would recognize, such as variables and conditional statements.

Space in this book doesn't allow for a detailed description of all of the many features of BPEL, but in this section we'll highlight the main structure of a BPEL document. This will help us describe how these features are mapped to the familiar SCA concepts of component services, references, and properties and hence how a BPEL process can be used to implement an SCA component.

Listing 6.4 shows the outline of a BPEL process used to implement the Payment component. As before, this component is expected to take payments from customers. To achieve this, the Payment component must arrange for a credit card payment to be taken and then must send an email to the customer confirming that the payment has

been taken. This is becoming a familiar scenario now, and it's a simple process when expressed in BPEL.

Listing 6.4 The outline of a BPEL process document

```
<process name="Payment" ...>
  <import location="Payment.wsdl" .../>
  <import location="CreditCardPayment.wsdl" .../>
  <import location="EmailGateway.wsdl" .../>

   <partnerLinks>
      <partnerLink name="paymentProcessPartnerLink" .../>
      <partnerLink name="creditCardPaymentPartnerLink" .../>
      <partnerLink name="emailGatewayPartnerLink" .../>
   </partnerLinks>

   <variables>
     <variable name="makePaymentRequestMessage"
               messageType="pp:MakePaymentRequest"/>
     ...
   </variables>

   <sequence>
       <receive name="start" .../>

       <assign name="createCreditCardRequest">
         ...
       </assign>

       <invoke name="invokeCreditCardPayment" .../>

       <assign name="createEmailGatewayRequest">
          ...
       </assign>

       <invoke name="invokeEmailGateway" ... />

       <assign name="createResponse">
           ...
       </assign>

       <reply name="end" .../>
   </sequence>
</process>
```

Some of the detail has been omitted in order to clearly show how the main elements in the BPEL process appear. The `<partnerLink>` elements describe how the process interacts with other services, the `<variable>` elements hold data as the process progresses, and the `<sequence>` element holds the activities that describe the steps in the business process.

PARTNER LINKS

To be useful, a BPEL process has to be able to call and be called by other services or partners. Partner links describe connections with other services in terms of WSDL 1.1 port types. A partner link can describe the roles of both communicating parties and

therefore can be associated with two roles. `myRole` refers to the role that the BPEL process plays, and `partnerRole` refers to the role that the partner plays in any interaction.

VARIABLES

As a business process continues, state can be accumulated using variables. Variables can, for example, hold incoming message data or the data that results from calling other services. Variables can be further transformed and assigned to other variables and then used as the input to other services.

ACTIVITIES

The main building blocks of a BPEL process are activities. A number of activities are defined by BPEL, including

- `<receive>`—Wait for a message to arrive
- `<invoke>`—To invoke a web service
- `<if>`—Branch based on an expression

Activities are strung together within a BPEL `<sequence>` and can be nested in various ways in order to represent decisions, concurrency, and asynchronicity in a business process.

Much more detailed information can be found by looking at the OASIS WSBPEL specifications, but this should provide enough of a refresher in order for you to understand how a BPEL process can be used to implement an SCA component. Let's carry on now and look at just that.

6.2.2 BPEL in Tuscany and SCA

From the SCA point of view, a BPEL process defines a component type that maps partner links to SCA component services and references. The sample that demonstrates how this is done can be found in the contributions/payment-bpel and contributions/payment-bpel-process directories of the travel-booking sample.

Figure 6.5 shows an overview of how the mapping takes place.

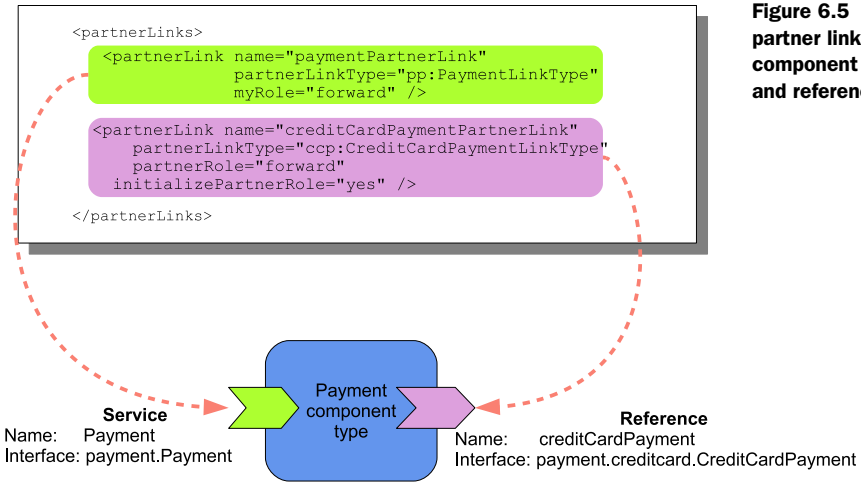


Figure 6.5 Mapping partner links to SCA component services and references

The SCA specification also maps the variables of a BPEL process to SCA properties, although the Tuscany runtime doesn't support this at the time of writing.

For the payment process example, the following listing shows the component type derived from the partner links in the BPEL process shown in figure 6.4.

Listing 6.5 The component type derived from the payment BPEL process

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <service name="paymentProcessPartnerLink">
    <interface.wsdl interface=
     "http://www.example.org/PaymentProcess/#wsdl.interface(PaymentProcess)"/>
  </service>

  <reference name="creditCardPaymentPartnerLink">
    <interface.wsdl interface=
      "http://.../CreditCardPayment/#wsdl.interface(CreditCardPayment)"/>
  </reference>

  <reference name="emailGatewayPartnerLink">
    <interface.wsdl interface=
      "http://www.example.org/EmailGateway/#wsdl.interface(EmailGateway)"/>
  </reference>

</componentType>
```

Note that the service and reference names map directly to the names of the partner links as they appear in the BPEL process. Figure 6.6 gives a pictorial overview of how the BPEL partner links from our payment process resolve to SCA services and references.

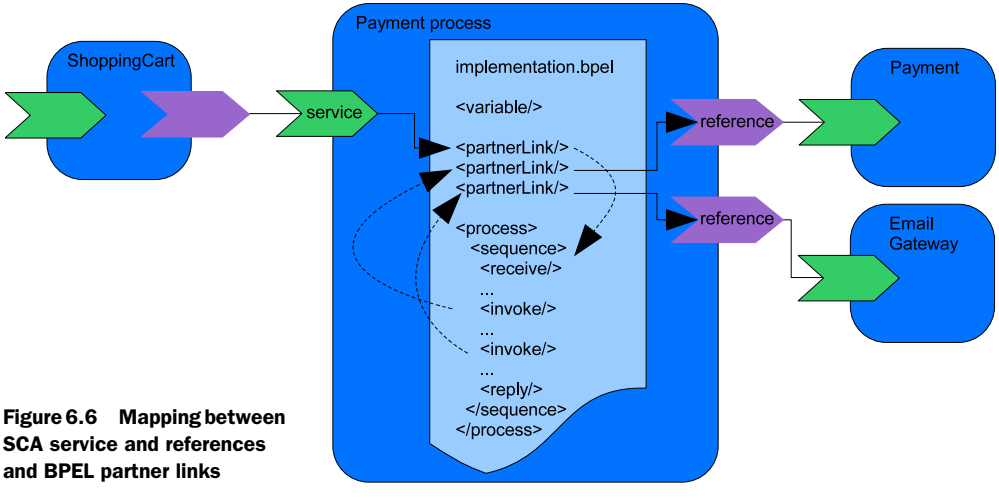


Figure 6.6 Mapping between SCA service and references and BPEL partner links

The arrows in figure 6.6 show the flow of control from the SCA service, through the BPEL process partner links and activities, and out through SCA references.

A message arrives, via an SCA binding, at an SCA service. The SCA service equates to a BPEL partner link. In this case, the incoming message gives rise to the BPEL `<receive>` action executing. Within the `<receive>` action, two `<invoke>` actions create messages that are sent out via related partner links. These partner links relate to SCA references, and hence the messages are sent out over SCA bindings to the target services configured in the composite file.

The BPEL concept of a partner link corresponds to both SCA services and references. In BPEL a partner link is designed to describe the relationship between two services where both services can originate messages. A partner link type can include two roles to describe the role at each end of the link and what interface the role implies.

Partner links themselves explicitly describe `myRole` and `partnerRole` with reference to the WS-BPEL process in question. Figure 6.7 shows how these roles map to the interfaces of services and references.

If you're at the service end, then the forward interface describes the interface that the BPEL process exposes and hence maps to `myRole`, the role that the WS-BPEL process is playing. The `partnerRole` then describes the role that the calling service plays and the interface that it exposes. In SCA terminology, this is the callback interface.

At the reference end it's the other way round, where the forward interface corresponds to the partner role as the BPEL process calls the partner service. The `myRole` for the WS-BPEL process in this case is to provide a callback interface. Note, though, that Tuscany doesn't yet support callbacks for components implemented using implementation.bpel.

Partner link roles map to WSDL port types. Hence, the forward and callback interfaces that a partner link represents are clearly described in a language that the Tuscany runtime understands. The Tuscany runtime is able to read the WSDL descriptions associated with partner links, and no special mapping is required to interpret the interfaces that BPEL scripts provide or require.

Let's look in detail at how BPEL partner links map to SCA services.

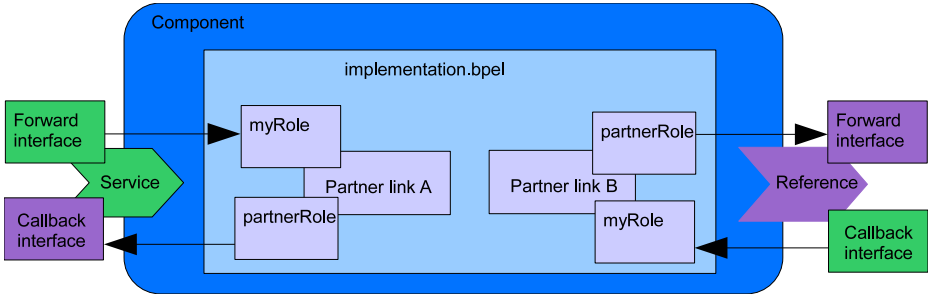


Figure 6.7 The mapping between partner link roles and SCA interfaces on services and references

6.2.3 Mapping WS-BPEL partner links to SCA services

The SCA Client and Implementation Model Specification for BPEL describes in detail the set of rules that should be applied for identifying the services and references for BPEL processes. SCA services map to those partner links whose roles are set such that the BPEL process is the receiver of a message. All other partner links map to SCA references.

The payment BPEL process can be found in the payment.bpel file from the `payment-bpel-process` contribution. Looking at this BPEL process in more detail, you can see that the process is initiated by a `<receive>` action:

```
<receive name="start"
         partnerLink="paymentProcessPartnerLink"
         portType="pp:PaymentProcess"
         operation="makePayment"
         variable="makePaymentRequestMessage"
         createInstance="yes"/>
```

Clearly this action waits to receive a message. The `paymentProcessPartnerLink` is referred to here, and the forward interface that this partner link specifies will be a service interface for this process.

A BPEL process can also be started by `<onMessage>` and `<onEvent>` elements. These result in the BPEL process taking some action based on an incoming message, and so partner links associated with these elements will also map to SCA services.

Looking in more detail at the `paymentProcessPartnerLink`, you can see how it's defined within the BPEL process:

```
<partnerLink name="paymentProcessPartnerLink"
             partnerLinkType="pp:PaymentProcessLinkType"
             myRole="forward" />
```

The partner link type `pp:PaymentProcessLinkType` defines the forward interface for this partner link. As we said, in this scenario, this partner link defines the service interface that the BPEL process exposes. Typically, the partner link type will be defined in the WSDL file for the payment process, Payment.wsdl in this case, in the following way:

```
<plnk:partnerLinkType name="PaymentProcessLinkType">
    <plnk:role name="forward" portType="tns:PaymentProcess"/>
</plnk:partnerLinkType>
```

The partner link type refers to a WSDL port type called `tns:PaymentProcess`, and hence the interface contract for the service that the BPEL process exposes can be established.

Note that in this case the partner link type defines only a single role named `forward`. Notice also that the `myRole="forward"` attribute of the partner link matches the defined role on the partner link type. In partner link terminology, `myRole` is the role of the BPEL process, and this specifies that the BPEL process provides a service with the interface defined by the port type `tns:PaymentProcess`. With a single role,

this partner link defines a forward-only service interface. If a `partnerRole` were defined, then this would describe a callback interface that the partner provides.

Looking back, you see that the `<receive>` action exploits the same port type as the `forward` role. Hence, when it comes time to specify the SCA service, you see that the interface refers to the same `PaymentProcess` port type:

```
<service name="paymentProcessPartnerLink">
  <interface.wsdl interface=
   "http://www.example.org/PaymentProcess/#wsdl.interface(PaymentProcess)"/>
</service>
```

Note that the names of the services and references that you specify in the composite file must match the names of the partner links that appear in the BPEL process.

The Tuscany runtime is able to introspect the BPEL process and generate a component-type model based on the partner links that it finds. Any services and references that are identified can then be configured with information, such as binding configuration, in the composite file.

Partner links that describe the role of the BPEL process as the receiver of a message are mapped to services. Any other partner links are mapped to references, so let's look at references in a little more detail.

6.2.4 Mapping WS-BPEL partner links to SCA references

Any partner links that aren't identified as services map to SCA references. In the case of the payment process, let's look at the `<invoke>` elements and the partner links that they use. A good example is the `invoke` element that calls the EmailGateway service.

```
<invoke name="invokeEmailGateway"
        operation="sendEmail"
        inputVariable="sendEmailRequestMessage"
        outputVariable="sendEmailResponseMessage"
        partnerLink="emailGatewayPartnerLink"
        portType="eg:EmailGateway" />
```

Note that a partner link is referenced. In this case it's the `emailGatewayPartnerLink`, which is defined in the following way:

```
<partnerLink name="emailGatewayPartnerLink"
             partnerLinkType="eg:EmailGatewayLinkType"
             partnerRole="forward"
             initializePartnerRole="yes" />
```

The partner link type in this case is as follows:

```
<plnk:partnerLinkType name="EmailGatewayLinkType">
    <plnk:role name="forward" portType="tns:EmailGateway"/>
</plnk:partnerLinkType>
```

Again the partner link has only one role, and it defines the forward interface between the payment process and the email gateway. In this case, the forward role is defined as being the `partnerRole`. The partner role refers to the partner with which the BPEL

process is interacting, the email gateway service in this case. The `EmailGateway` port type is associated with the forward role, so this is the port type that the email gateway service provides and that the reference to the email gateway will use.

In the composite file we can define a reference with the same name as the partner link and define the reference's interface and bindings in the normal way.

```
<reference name="emailGatewayPartnerLink">
  <interface.wsdl interface=
     "http://www.example.org/EmailGateway/#wsdl.interface(EmailGateway)"/>
</reference>
```

This reference has the default multiplicity of `1..1` and hence can only be connected to a single service. The SCA Client and Implementation Model Specification for WS-BPEL describes the set of rules that determine how references with other multiplicities are identified, but these depend on SCA extensions being included in the BPEL script, and these aren't yet supported by the Tuscany runtime.

6.2.5 Handling errors

BPEL defines an extensive set of mechanisms for catching and throwing business and systems faults during the execution of a business process. If you look at the SCA Client and Implementation Model Specification for WS-BPEL, you'll note that it doesn't describe how faults are dealt with.

Currently the Apache Tuscany runtime will return faults thrown by the BPEL script but won't pass faults through references that are returned by services that the WS-BPEL script itself calls. Lifting this limitation would be good place to start for those who are interested in BPEL and who are also interested in contributing to the Tuscany code base.

6.2.6 Limitations of implementation.bpel in Tuscany 1.x

The SCA Client and Implementation Model Specification for WS-BPEL defines a set of BPEL extensions. None of these extensions are yet supported by the Tuscany runtime. The lack of support for these extensions means that the following features aren't supported:

- Properties
- Multivalued references

The Tuscany runtime also lacks support for the following features:

- interface.partnerLinkType
- Local partner links
- Conversations
- Callbacks
- Faults across references
- Implementation policies

It's hoped that these restrictions will be lifted as more features are added to the implementation.bpel module in Tuscany.

Locating the Apache ODE runtime database

Tuscany relies on the Apache ODE BPEL engine to provide support for running BPEL scripts. The ODE engine itself relies on a Derby database being in place, and in our samples you'll see that Maven is configured, through a `maven-dependency-plugin` target, to unpack the following artifact:

```
<groupId>org.apache.ode</groupId>
<artifactId>ode-dao-jpa-ojpa-derby</artifactId>
```

The result of this is that, when run from Maven, each sample has a Derby database automatically added to its \target\test-classes\jpadb directory before the test cases are run. Hence, the ODE engine is able to find the database that it requires.

Now that you've seen how WS-BPEL can be exploited from within an SCA composite application, let's move on and look at how scripts can similarly be used to implement SCA components.

6.3 Implementing components using scripts

The SCA specifications don't yet define any implementation types that allow scripting languages to be used to provide component implementations. Tuscany does provide an extension that exploits the Apache Bean Scripting Framework (BSF) to provide a scripting implementation type. You can implement components using BSF-compliant scripts by using the `<implementation.script>` element. Feel free to skip this section if you're not planning to use scripts in your SCA application.

6.3.1 BSF-based script implementations in Tuscany and SCA

BSF is a subproject of the Apache Jakarta project (http://jakarta.apache.org/bsf/) and is designed to provide a generic interface for embedding various types of scripting engines, such as Python (using Jython), Ruby, and Groovy, into Java applications.

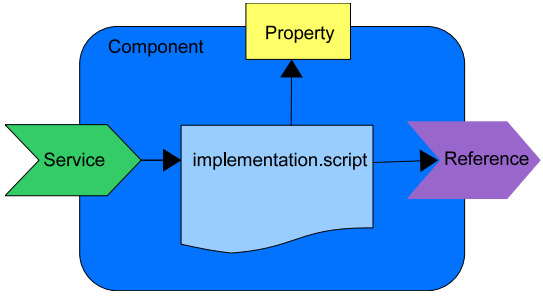


Figure 6.8 Implementation.script allows SCA components to be implemented using those scripting languages supported by the Bean Scripting Framework from the Apache Jakarta project.

Therefore, by using implementation.script you can build component implementations using any of the scripting languages that BSF 3 supports. Figure 6.8 shows how the implementation.script type provides an SCA component implementation.

Again we'll use the Payment component example to show how a script can be used for a component implementation. We'll use Groovy as the scripting language of choice here to show how services, references, and properties are mapped. The principles applied here apply to the other scripting languages supported by BSF. You can find the sample in contribution/contributions/payment-groovy.

If you want to see one of the other script environments supported by BSF in action, we've also provided a Python version of the Payment component. You can find this in the `payment-python` contribution. It's similar to the Groovy version, so we'll stick with Groovy for following discussion.

Following is the Groovy script that provides the Payment component implementation. This script can be found in the file payment/PaymentImpl.groovy.

Listing 6.6 A Groovy script payment component implementation

```
def makePaymentMember(customerId, amount) {

    def finalAmount = amount + transactionFee;

    scatours.emailgateway.EmailType email =
         new scatours.emailgateway.EmailType();
    email.setTo(customerId);
    email.setTitle("Payment " + finalAmount +  " Received");

    emailGateway.sendEmail(email);

    return "Success";
}
```

In this example we'll show the `makePaymentMember()` operation, where the customer is expected to already be a registered member of the TuscanySCATours website. The call to the credit card details component is omitted in this case. The code shows the call being made to the email gateway component. The component is defined in the payment composite file in the following way:

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           name="payment">
  <component name="PaymentComponent">
    <tuscany:implementation.script script="payment/PaymentImpl.groovy"/>
    <service name="Payment">
       <interface.java interface="payment.Payment" />
```

```
    </service>
    <reference name="emailGateway" target="EmailGatewayComponent">
       <interface.java interface="scatours.emailgateway.EmailGateway" />
    </reference>
    <property name="transactionFee" type="xsd:float">9.73</property>
  </component>
<composite/>
```

Note that `<implementation.script>` is in the Tuscany namespace because it's not defined in the SCA specifications.

6.3.2 Defining interfaces for script-based SCA services and references

Scripting languages tend to be dynamically typed and generally don't provide a mechanism for statically describing the interfaces exposed by the services they provide or for the references they consume.

Apache Tuscany exploits the information provided by the developer through the composite file and through a component type file, if present, to determine the type of the various interfaces that a script uses.

If interface information isn't available through one of these two files, then the Tuscany runtime assumes that a dynamic invocation is required. In this case, it takes the operation name from the incoming message and invokes BSF. In this way, script writers can either construct a component type file manually to explicitly identify services, references, and properties or rely on the information in the script and the composite file to form the component type. This approach for determining the interfaces in use is the same regardless of which scripting language is used.

6.3.3 Mapping between SCA services and scripts

In general, each operation declared in the script maps to a service defined on the component in the composite file. In our payment example, the Groovy script declares an operation called `makePaymentMember`. In the composite file, the service is defined in the usual way:

```
<service name="Payment">
    <interface.java interface="payment.Payment" />
</service>
```

The interface is defined using the Java `payment.Payment` interface. The Tuscany runtime introspects the Java `payment.Payment` interface and finds the `makePaymentMember` operation. The runtime then uses BSF to locate the `makePaymentMember` in the Groovy script. When a message arrives, Tuscany is able to direct the message to the correct function in the script and uses BSF to load the script and run that operation.

Because scripts tend to be dynamically typed, Tuscany relies on the interface information provided through the composite file or optionally through the component type file in order to perform appropriate transformations on messages arriving from a service binding.

6.3.4 Mapping between SCA references and scripts

A similar approach works for references. You'll note that there's no annotation in the Groovy script that identifies references. The Tuscany runtime injects fields into the script context based on the references it finds in the component definition. In this case, the `emailGateway` reference is injected:

```
<reference name="emailGateway" target="EmailGatewayComponent">
   <interface.java interface="scatours.emailgateway.EmailGateway" />
</reference>
```

Again a Java interface is used to describe the reference interface. This reference is now available in the script environment and is used as follows:

```
emailGateway.sendEmail(email);
```

The script writer didn't explicitly define the `emailGateway` variable but used it in the knowledge that the Tuscany runtime will inject it automatically. References are differentiated from properties based on the configuration that appears in the composite file.

6.3.5 Mapping between SCA properties and scripts

The property definitions that appear in the composite file are used by the runtime to inject property values into the script context:

```
<property name="transactionFee" type="xsd:float">9.73</property>
```

The runtime then makes a property, called `transactionFee`, available in the script without the script having to define the property. You can see in our example that the processing charge is added to the amount being charged:

```
def finalAmount = amount + transactionFee;
```

You can define properties in the script itself if you need to use the script outside the context of the SCA component.

The automatic insertion of references and properties may look a little odd because the script developer hasn't defined these variables explicitly. You may be wondering how they know what variable names to use, but implementation.script has been designed with the expectation that the script developer will develop the script in conjunction with the component description. This gives the developer an easy way of associating any of the Tuscany bindings with a script without any script-specific code being required.

6.3.6 Handling errors

If the Bean Scripting Framework experiences a problem in executing a script that it's hosting, then, in theory, it will throw a `ScriptException`. Currently the Tuscany runtime is set up to catch this exception and add it to the response message. At the time of writing, however, there's an issue with the runtime in that BSF isn't throwing exceptions when it should.

Improving the way that script errors are handled would be a good project for those interested in contributing to the Tuscany and possibly the BSF code bases.

6.4 Summary

In the early chapters of the book you learned how SCA offers a technology-agnostic component model. Its support of various component implementation types enables different technologies to work with one another in an SCA composite application. Two components implemented using different technologies can communicate with each other across any of the Tuscany bindings.

This chapter has offered a technical introduction to three of the many component implementation types that Tuscany supports today. By now, you've learned how to implement components using Spring, BPEL, and scripting languages. This provides a good demonstration of the flexibility of SCA to embrace implementation technologies.

Tuscany has more implementation types in development, for example, EJB, Java EE, and Web, and the list is growing as the community contributes new implementation types by leveraging Tuscany's extensible architecture. Look forward to chapter 14 if you want to know how new implementation types can be added using Tuscany's implementation extensibility point.

Now that you have a good grounding in implementing SCA components, let's move on and explore how you can connect components using different communication protocols by configuring SCA bindings.

7 Connecting components using bindings

This chapter covers

- Connecting services and references using SCA bindings
- Exposing SCA services using Web Services, CORBA, RMI, and JMS
- Integrating services exposed using Web Services, CORBA, RMI, JMS, or as EJBs from SCA

By this stage in the book you know how to implement SCA components and wire them together to create SCA applications. It's now time for you to shift your focus and look outside the world of SCA and see how you can integrate your SCA applications with the rest of the world.

Often an SCA application will need to expose its services to existing business applications or make use of the services offered by them. This is done over a communications channel and protocol that's understood by both parties. These existing business applications typically make use of a variety of communications protocols such as SOAP, CORBA, RMI, JMS, and so on.

This chapter is all about SCA bindings. You'll see how SCA hides some of the complexities of handling communications protocols from the SCA application developer using SCA bindings. You'll also see how SCA bindings allow the communications protocols to be changed with little effort.

This chapter uses components from the SCA Tours application to show how you can make services available to non-SCA applications using Tuscany SCA bindings. We'll also demonstrate how other applications can expose their services to SCA components so their services can be invoked using Tuscany SCA bindings.

This chapter is about the bindings that Tuscany supports. Some of these are described by the SCA specification, such as

- Web Services binding (SOAP)
- JMS binding
- EJB session binding

Tuscany also supports additional bindings that have been defined based on the Tuscany community's demand, such as

- CORBA binding
- RMI binding

By the end of this chapter, you'll know how to integrate SCA services and references with the wider world of non-SCA applications using SCA bindings. Let's start with a quick review of SCA bindings in general.

7.1 Introduction to SCA bindings

SCA supports many communications protocols. In SCA terms, these are called *bindings*. Bindings encapsulate the complexities of the communications protocols and enable components to be implemented and wired together without a direct dependency on the communications protocol(s) used.

What does this mean in reality? It means that an SCA developer can implement a component without polluting the business code with details of how the communication between components will be handled. This separation of concerns has the added advantage of making the communications protocol a pluggable entity that can be changed at any time based on how and where the component is deployed. For example, suppose you want to make the TuscanySCATours currency converter a web service. Simple—you add the SCA Web Services binding to the service defined in the composite XML file. Suppose you then want to make the TuscanySCATours currency converter an RMI service as well. Simple—you add the SCA RMI binding to the service defined in the composite XML file. You can do both of these changes without modifying the component implementation code for the currency converter. We'll cover both of these examples later in this chapter.

As discussed in section 2.5.1, SCA bindings can be added to a composite application either on an SCA service or on an SCA reference. The following subsections provide a quick overview of these two usage patterns. We'll discuss specific bindings in more detail later in this chapter.

7.1.1 Using SCA bindings on an SCA service

One or more SCA bindings can be used on a service to allow the service to be invoked over each binding's protocol. To illustrate this, listing 7.1 shows the Payment composite with an SCA Web Services binding and an SCA RMI binding on the Payment service.

Listing 7.1 Payment composite XML file with Web Services and RMI bindings

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com"
           name="payment">
    <component name="Payment">
        <implementation.java class=
            "com.tuscanytours.payment.impl.PaymentImpl" />
        <service name="Payment">
            <interface.java
                interface="com.tuscanyscatours.payment.Payment" />
            <binding.ws />
            <binding.rmi />
        </service>
    </component>
</composite>
```

SCA Web Services binding (1)

SCA RMI binding (2)

By adding the Web Services binding (1) and the RMI binding (2), the Payment service can be invoked over both SOAP and RMI.

7.1.2 Using SCA bindings on an SCA reference

SCA references describe the dependencies that an SCA component has on other services. Multiple bindings can be assigned to a single reference, each enabling communication over a different type of protocol. Listing 7.2 shows how the Payment component can invoke the CustomerRegistry service over the Web Services binding and CreditCardCORBAService over the CORBA binding.

Listing 7.2 Payment composite with various SCA bindings on the references

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com"
           name="payment">
   <component name="Payment">
      <implementation.java class=
          "com.tuscanyscatous.payment.impl.PaymentImpl" />
      <reference name="customerRegistry">
          <binding.ws
             uri="http://someserver/CustomerRegistry"/>
      </reference>
      <reference name="creditCardPayment">
          <binding.corba
            host="someserver" port="5080"
            name="CreditCardCORBAService"/>
      </reference>
      <reference name="emailGateway"
                 target="EmailGatewayComponent"/>
  </component>
</composite>
```

Web Services binding (1)

CORBA binding (2)

The `customerRegistry` reference on the Payment component has a Web Services binding ❶. This is indicated by the `<binding.ws>` element, which specifies the location of the customer registry web service using the `uri` attribute.

The `creditCardPayment` reference on the Payment component has a CORBA binding ❷. This is indicated by the `<binding.corba>` element, which specifies the host and port number as well as the CORBA service name of the card payment service.

Looking at the `emailGateway` reference, did you notice that no binding is listed? So which SCA binding is used for this reference? The answer is that if a service or reference doesn't specify a binding, then the SCA default binding is used.

Before we move on to look at some SCA bindings, let's spend a few moments introducing the SCA components we'll use to demonstrate the SCA bindings.

7.2 Demonstrating SCA bindings

Throughout this chapter, we'll demonstrate how to use bindings on services with the CurrencyConverter component and on references with the Notification component. Let's have a quick look at the CurrencyConverter and Notification components before we move on to look at specific bindings.

7.2.1 Overview of the currency converter

The CurrencyConverter is a simple component that can be used to calculate the value of one currency in a different currency. For example, it can be used to calculate the value of $100 U.S. in sterling. Figure 7.1 shows an overview of the currency converter.

The composite definition for the currency converter SCA component can be found in the contributions/currency directory of the SCA Tours application and is shown here.

Listing 7.3 The currency converter composite XML file

```
<composite name="currency-converter" ...>
    <component name="currencyconverter">
        <implementation.java class= "com.tuscanyscatours.
        ➥ currencyconverter.impl.CurrencyConverterImpl" />
        <service name="CurrencyConverter">
            <interface.java interface= "com.tuscanyscatours.
            ➥ currencyconverter.CurrencyConverter" />
        </service>
    </component>
</component>
```

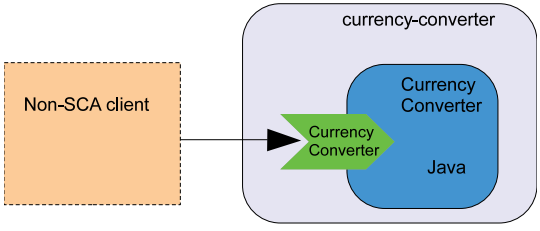


Figure 7.1 The currency converter with its single CurrencyConverter service

As you can see from the XML in listing 7.3, the currency converter is implemented using the Java implementation type. It exposes a single service that's defined by the `CurrencyConverter` Java interface, as shown in the following listing.

Listing 7.4 The currency converter Java interface

```
@Remotable
public interface CurrencyConverter {
    double getExchangeRate(String fromCurrencyCode,
                           String toCurrencyCode);
    double convert(String fromCurrencyCode, String toCurrencyCode,
                   double amount);
}
```

Listing 7.4 shows the Java interface that defines the methods on the currency converter service. It has two methods, `getExchangeRate` and `convert`.

The CurrencyConverterLauncher in the launchers/currency-converter directory of the SCA Tours application can be used to test the currency converter. When it's run, it will load the currency converter contribution into Apache Tuscany and perform a quick test, producing the following output:

```
Quick currency converter test
USD -> GBP = 0.5
100 USD = 50.0GBP
```

Because we haven't specified any bindings on the services or references, Tuscany will use the SCA default binding to wire the components together.

7.2.2 Overview of the Notification service

The Notification service is a simple component that can be used as part of an SCA application to send notifications to users. For the SCA Tours application, it could be used to send notifications such as "credit card payment accepted." The Notification service acts as a façade that hides how the notification is sent to the user. In this basic implementation, the Notification service sends notifications via SMS to the user's mobile phone. The Notification service doesn't contain the code to send the SMS. To send the SMS, it invokes an external SMS gateway service. Figure 7.2 shows an overview of the Notification service.

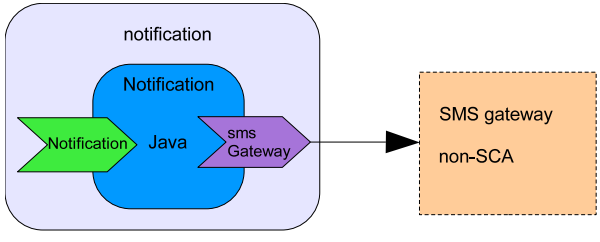


Figure 7.2 Invoking the Notification service will cause it to invoke the SMS gateway web service.

The composite definition for the Notification component is shown in here.

Listing 7.5 The notification composite XML file

```
<composite name="notification" ...>
    <component name="Notification">
        <implementation.java class="com.tuscanyscatours.notification.
        ➥ impl.NotificationImpl" />
        <service name="Notification">
            <interface.java  interface="com.tuscanyscatours.
            ➥ notification.Notification"/>
        </service>

        <reference name="smsGateway">
            <interface.java
               interface="com.tuscanyscatours.smsgateway.SMSGateway"/>
            ...
        </reference>
    </component>
</composite>
```

The Notification service is implemented using the Java implementation type. It exposes a single service that's defined by the `Notification` Java interface. It also has a reference to the SMS gateway.

The `Notification` Java interface is used to define the methods on the Notification service:

```
public interface Notification {
    boolean notify(String accountID, String subject, String message);
}
```

It has a single method called `notify` that will

- Look up the mobile phone number associated with the specified account ID
- Send an SMS to the user's mobile phone that contains the specified subject and message

The Notification service assumes that there is a mobile phone number associated with the specified account ID.

The SMS gateway is a non-SCA application that exposes a service that can be used to send SMS messages to mobile phones. It represents an existing business application, not written using SCA, which we'll want to integrate into our SCA application.

The interface of the service that's exposed by the SMS gateway is shown here:

```
public interface SMSGateway {
    boolean sendSMS(String fromNumber, String toNumber, String text);
}
```

The `SMSGateway` Java interface for the SMS gateway has a single method called `sendSMS` that will send an SMS message containing the given text to the specified number. We won't really send SMS messages from our sample code, but we'll write the message that would be sent to the console.

Now that you have an overview of the currency converter and Notification components, let's move on and look at our first binding, the SCA default binding.

7.3 Connecting component services with binding.sca

As you've seen, an SCA binding is required for communication between components. You saw how you can assign a binding to a service or a reference. It's also possible to omit the binding, which means Tuscany will use the default SCA binding. You can also tell Tuscany to use a default binding explicitly by using the `<binding.sca/>` element on a reference or service.

It's expected that for most wires between SCA components, no binding will be specified because the default SCA binding will automatically be used. After all, most of the time a developer doesn't care how two SCA components are physically connected as long as one can invoke the other. It's only mandatory to specify a binding when you want to connect an SCA component to an application running outside the SCA domain.

The default SCA binding is defined by the SCA Assembly specification as a binding that can be used to wire together components within a single SCA domain. It's not designed to wire together components that exist in multiple SCA domains or are external to the SCA domain. If a service needs to be invoked from outside the SCA domain, or a reference needs to invoke a service outside the SCA domain, then an interoperable binding (such as the Web Services or JMS binding) must be used. We'll move on in a moment to look at some of these interoperable SCA bindings that Tuscany provides.

In Tuscany, the communications protocol used by the default binding will depend on whether the service and reference are in the same node or in different nodes. If they're in different nodes, Tuscany will use SOAP/HTTP for the default binding. If they're in the same node, Tuscany will use Java method invocations via a Java proxy because the target will always be in the same JVM. This use of proxy calls within the same node happens with both local and remotable interfaces. The composite XML in the next listing shows how the default SCA binding can be used.

Listing 7.6 Payment composite with various usages of the SCA default binding

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com"
           name="payment">
  <component name="Payment">
      <implementation.java class=
          "com.tuscanyscatours.payment.impl.PaymentImpl" />
      <reference name="customerRegistry"
                 target="CustomerRegistry"/>
      <reference name="creditCardPayment"
                 target="CreditCardPayment">
          <binding.sca/>
      </reference>
      <reference name="emailGateway">
```

1 Implicit SCA default binding

2 Explicit SCA default binding

```
            <binding.ws
               uri="http://someserver/EmailGateway"/>
            <binding.sca/>
         </reference>
   </component>
</composite>
```

❸ **Explicit Web Services and default bindings**

The `customerRegistry` reference ❶ doesn't list any bindings, so the SCA default binding will be used. The `creditCardPayment` reference explicitly specifies the SCA default binding ❷. This isn't strictly necessary because it will automatically be added by Tuscany because there are no other bindings specified. The `emailGateway` reference has an SCA Web Services binding and an SCA default binding ❸. For this reference, if the SCA default binding wasn't explicitly listed, it wouldn't be automatically added by Tuscany because the reference already has an SCA Web Services binding.

In the previous example, we mentioned the SCA Web Services binding, `binding.ws`, so let's move on and have a look at it now.

7.4 Connecting component services with web services

In this section, we'll look at how the SCA Web Services binding, `binding.ws`, can be used to expose SCA services as web services and how you can access web services via SCA references.

What are web services?

The term *web service* is generally used to describe an interface that's provided by a software application that's callable over the network. More recently, the term has come to be used to specifically describe services provided over a network using the SOAP over HTTP protocol. SOAP started life as an acronym for Simple Object Access Protocol (some say Service Oriented Access Protocol) and describes an XML-based format for passing messages from a client to a service and back. The XML is simple and defines an envelope that holds a body, the message itself, and a header that contains metadata about the message. Running SOAP over HTTP means POSTing SOAP-formatted XML to an HTTP server where special software, such as Apache Axis, reads the XML and calls the right service operation with data from the message.

Several libraries available today allow a developer to author and consume SOAP/HTTP web services. Many of them require you to either code to the library's API or use a tool to generate service stubs and client proxies that hide the library API from you. Tools are often also provided for tasks like generating WSDL descriptions from service interfaces. Such tools are useful, but Tuscany makes SOAP/HTTP even easier to use. The Tuscany component developer adds the `<binding.ws>` element to a component's services or references, and the job's done. It's worth mentioning that Tuscany 1.x implements the

OSOA SCA specifications that are based on WSDL 1.1. Tuscany 1.x uses Apache Axis 2 as its SOAP stack.

Let's start by looking at using `binding.ws` on a component service.

7.4.1 Exposing an SCA service as a web service

Let's first look at how to expose an SCA component as a web service using the SCA Web Services binding and then create a web service client to access it. We'll be adding the `<binding.ws>` element to the composite XML file.

USING THE SCA WEB SERVICES BINDING ON A SERVICE

To expose the currency converter as a web service, you'll need to add the SCA Web Services binding to the `<service>` element in the composite XML file. The code for this can be found in the contributions/currency-ws directory of the SCA Tours application and is shown in the following listing.

Listing 7.7 Currency converter component with SCA Web Services binding

```
<component name="CurrencyConverter">
    <implementation.java
       class="com.tuscanyscatours.currencyconverter.impl.
       ➥ CurrencyConverterImpl" />
    <service name="CurrencyConverter">
        <binding.ws />
    </service>
</component>
```

Listing 7.7 shows the currency converter component XML with the addition of the SCA Web Services binding. But wait a minute; you may wonder if we've really turned the currency converter into a web service by adding one line to an XML file. The answer is yes; let's run the example to demonstrate this.

RUNNING THE CURRENCY CONVERTER WEB SERVICE LAUNCHER

The currency converter web service launcher (CurrencyConverterWSLauncher) in the launchers/currency-converter-ws directory of the SCA Tours application can be used to test the currency converter using the Web Services binding. When it's run directly (or via Ant run), it produces the same output as previous runs of the currency converter. But the currency converter now has a web service interface. How can you verify that it does? You could write a web service client to access it. You'll be doing that later, but there's an easier way using a web browser. When the Web Services binding is used with a service, it generates and publishes a description of the service using WSDL (Web Services Definition Language.) To view the currency converter WSDL while it's running, point your browser at the following URL: http://localhost:8080/CurrencyConverter?wsdl. In Firefox, the WSDL will be displayed. In Safari, you must use the View > View Source menu to see the WSDL.

What is WSDL?

WSDL stands for Web Services Definition Language and is an XML language that can be used to describe the interfaces provided by a web service. It allows the functionality of an interface of a web service to be described in terms of the operations it provides and the physical details of where the web service is hosted.

The following listing shows a fragment of the WSDL document for the currency converter. Because it's quite long, we've removed the less-relevant details to keep it short.

Listing 7.8 Part of the WSDL for the currency converter service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="CurrencyConverterService"
  targetNamespace="http://currencyconverter.scatours/"
  xmlns:tns="http://currencyconverter.scatours/"
  ...
  <wsdl:binding name="CurrencyConverterBinding"
    type="tns:CurrencyConverter">
    <SOAP:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getExchangeRate">
      <SOAP:operation/>
      ...
    </wsdl:operation>
    <wsdl:operation name="convert">
      <SOAP:operation/>
      ...
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="CurrencyConverterService">
    <wsdl:port name="CurrencyConverterPort"
      binding="tns:CurrencyConverterBinding">
      <SOAP:address
        location="http://10.10.10.120:8080/CurrencyConverter"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

1 Web Services binding

2 Currency converter methods

3 Publish URL

The header of the WSDL document shows that this is the WSDL for our currency converter service. The WSDL specifies the SOAP document binding 1 and lists the `getExchangeRate` and `convert` methods 2 of the currency converter. The URL that's used to publish the currency converter 3 is also listed in the WSDL. The IP address in the URL is likely to be different in your WSDL because it will match the IP address of your computer.

ACCESSING THE CURRENCY CONVERTER SCA WEB SERVICE USING A JAVA CLIENT

Now that the currency converter is exposed as a web service, let's write a non–SCA client that will make use of it. The code for the non-SCA client can be found in the clients/currency-converter-ws-jaxws directory of the SCA Tours application. To simplify creating

the client, we'll use the `wsimport` tool from the Sun JDK version 1.6 or above to convert the currency converter WSDL into Java classes by using the following command:

```
wsimport -p com.tuscanyscatours.client
➥ http://localhost:8080/CurrencyConverter?wsdl
```

As an alternative to using Sun JDK 1.6, we could have installed Apache Axis2 and used the wsdl2java command that it provides to convert the WSDL into Java classes.

You can now use the generated Java classes to write an application to use the currency converter web service. The following code uses these generated Java classes to invoke the currency converter web service.

Listing 7.9 Fragment of the currency converter web service client

```
public static void main(String[] args) throws Exception {
  CurrencyConverterService service
    = new CurrencyConverterService();
  CurrencyConverter converter
    = service.getPort(CurrencyConverter.class);

  System.out.println("USD -> GBP = "
    + converter.getExchangeRate("USD", "GBP"));
  System.out.println("100 USD = "
    + converter.convert("USD", "GBP", 100.0) + "GBP");
}
```

The client uses the classes generated by the `wsimport` command to connect to the currency converter web service and does a quick test of the currency converter. Running this code, you'll see the following output:

```
USD-> GBP = 0.5
100 USD = 50.0 GBP
```

You've just invoked the currency converter service as a web service over the SCA Web Services binding.

One final point before we move on: You could have written the client to access the currency converter web service in a language other than the Java programming language. It would be relatively simple to use the WSDL produced by the Web Services binding to create clients in languages such as Microsoft C# or Visual Basic.

Now let's look at how to use SCA references to invoke services that are outside the SCA domain.

7.4.2 Accessing a web service using the SCA Web Services binding

It's possible to invoke an existing web service from an SCA application by using the SCA Web Services binding on a reference. It doesn't matter whether the web service is an SCA component or implemented in some other technology besides SCA.

CONNECTING AN SCA REFERENCE TO THE SMS GATEWAY WEB SERVICE

You'll now need to wire the SMS gateway reference in the Notification service to the web service exposed by the SMS gateway. This is done by adding the `<binding.ws>` element

to the reference. The code for this can be found in the contributions/notification-ws directory of the SCA Tours application and is shown in the following listing.

Listing 7.10 The SMS gateway reference with SCA Web Services binding

```
<composite name="notification" ...>
    <component name="Notification">
        <service name="Notification">
            ...
        </service>
        <reference name="smsGateway">
            <interface.java
                interface="com.tuscanyscatours.smsgateway.SMSGateway"/>
            <binding.ws
                uri="http://localhost:8081/SMSGatewayService"/>
        </reference>
    </component>
</composite>
```

When adding the Web Services binding to a reference, you'll need to tell Tuscany the URL of the web service using the `uri` attribute of the `<binding.ws>` element. For the SMS gateway, the URL is http://localhost:8081/SMSGatewayService.

RUNNING THE NOTIFICATION SERVICE CONNECTING TO THE SMS GATEWAY WEB SERVICE

Before you can run the Notification service, you'll need to ensure that the SMS gateway web service is running using the launcher in the services/smsgateway-jaxws directory of the SCA Tours application. Once it's running, you can run the Notification service using the launcher in the launchers/notification-ws directory of the SCA Tours application and connect to the web service of the SMS gateway. It will produce the following output on the console:

```
Quick notification test
Sending SMS to +44(0)7700900812 for accountID 1234
Node started - Press enter to shutdown.
```

The following output will also be displayed on the console where the SMS gateway was started:

```
Sending SMS message
From: +44(0)2079460723
To: +44(0)7700900812
Message: Holiday payment taken. Payment of £102.37 accepted...
```

As you can see from the two outputs, the Notification service has looked up the mobile phone number for account ID 1234 and used the web service interface of the SMS gateway to send the user an SMS.

Up to this point, you've used only the default configuration options for the Web Services binding. Let's see what other configuration options are supported.

7.4.3 Configuration options for the SCA Web Services binding

The Web Services binding supports several configuration options that allow you to customize how a service is exposed as a web service. The following listing shows the schema for the Web Services binding.

Listing 7.11 Schema for the SCA Web Services binding

```
<binding.ws
        name="NCName"
        uri="anyURI"
        wsdlElement="xs:anyURI"
        wsdlLocation="list of xs:anyURI"
        requires="sca:listOfQNames"
        policySets="sca:listOfQNames">
    <wsa:EndpointReference>
        ...
    </wsa:EndpointReference>
<binding.ws/>
```

Table 7.1 further describes the list of configuration options that are available for customizing a Web Services binding.

Table 7.1 Options for the SCA Web Services binding

Option	Optional	Description
`name`	Yes	The name for this binding.
`uri`	Yes	For services, the URI where the service should be published. For references, the URI of the service to connect to.
`wsdlElement`	Yes	The URI of a WSDL element.
`wsdlLocation`	Yes	The location of the WSDL document that describes this web service.
`requires`	Yes	Any required intents for this binding.
`policySets`	Yes	Any policy sets for this binding.
`EndpointReference`	Yes	WS-Addressing EndpointReference that specifies the endpoint for the service.

We'll briefly discuss the main usage scenarios of these options; see the OSOA SCA Web Services Binding specification for a complete description of all the usage scenarios. You can download it from http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

USING THE SCA WEB SERVICES BINDING WITH EXISTING WSDL DOCUMENTS

The Web Services binding makes extensive use of WSDL to describe the interfaces exposed by services and invoked by references. The Web Services binding can use an

existing WSDL document. This is often called WSDL-First because the WSDL document is created before the service is written. A common scenario for WSDL-First is where the SCA application needs to use an existing web service, such as in a cloud computing environment.

You can specify the WSDL document for a web service in a number of ways using the `wsdlElement` and `wsdlLocation` attributes of the Web Services binding. The simplest option is to use the `wsdlElement` attribute to refer to a specific service in a WSDL document, as illustrated in the following XML fragment:

```
<service name="CurrencyConverter">
    <interface.java interface="com.tuscanyscatours.currencyconverter.
    ➥ CurrencyConverter" />
    <binding.ws wsdlElement="http://localhost/CurrencyConverterService#
    ➥ wsdl.port(CurrencyConverter/CurrencyConverterSOAP)"/>
</service>
```

Another typical usage scenario is where the location of the WSDL document is specified using the `wsdlLocation` attribute, as illustrated in the following XML fragment:

```
<service name="CurrencyConverter">
    <interface.java interface="com.tuscanyscatours.currencyconverter.
    ➥ CurrencyConverter" />
    <binding.ws wsdlElement="http://localhost/CurrencyConverterService#
    ➥ wsdl.port(CurrencyConverter/CurrencyConverterSOAP)"
    ➥ wsdlLocation="http://localhost/CurrencyConverter.wsdl"/>
</service>
```

Both of these examples showed using the `wsdlElement` and `wsdlLocation` attributes of `binding.ws` on services. Note that they can both be used in the same manner for references.

USING THE SCA WEB SERVICES BINDING WITHOUT EXISITNG WSDL DOCUMENTS

The SCA web service can be used to expose a service without requiring a WSDL document to be provided. If a WSDL document isn't provided, then Tuscany will automatically generate a WSDL document from the Java interface of the service. This process is often called WSDL-Last because the WSDL document is produced after the implementation code.

The examples from earlier in this section used WSDL-Last because they used `<binding.ws>` without any of its attributes. The WSDL document can be retrieved by adding `?wsdl` to the end of the URL for the service. For example, if a web service is accessible at http://localhost:8080/CurrencyConverter, then its WSDL document can be retrieved from http://localhost:8080/CurrencyConverter?wsdl.

USING INTENTS AND POLICY SETS WITH THE SCA WEB SERVICES BINDING

Like all SCA bindings, the Web Services binding can have SCA intents and policy sets. As an example, the `SOAP.1_1` intent could be specified, and that would require the binding to use SOAP 1.1, as shown in the following XML fragment:

```
<service name="CurrencyConverter">
    <interface.java interface="com.tuscanyscatours.currencyconverter.
    ➥ CurrencyConverter" />
```

```
    <binding.ws requires="SOAP.1_1"/>
</service>
```

For more information on using SCA intents and policy sets, see chapter 10.

At this point, you know how to use the SCA Web Services binding on services and references. Let's move on and look at a different binding, the SCA CORBA binding.

7.5 Connecting component services with CORBA

In this section we'll look at how you can use the Tuscany SCA CORBA binding, `binding.corba`, to expose services as CORBA services and how you can invoke existing CORBA services from an SCA application. Feel free to read this section later if you're currently not planning to use CORBA in your SCA applications.

> **What is CORBA?**
>
> The Common Object Request Broker Architecture (CORBA) specification allows applications to communicate with each other over the network. It defines a standard protocol that allows programs that use CORBA to invoke each other regardless of the programming language they're written in or the operating system they're running. For example, a COBOL application running on Linux could expose a CORBA service, and it could be invoked by a Java application running on Microsoft Windows.

Tuscany makes it easy to call CORBA services. All you'll need to do is to add the `<binding.corba>` element to your component's services and references as appropriate. Let's start by looking at using `binding.corba` on a component service.

7.5.1 Exposing an SCA service as a CORBA service

In this section we'll show how the currency converter can be invoked as a CORBA service. See section 7.2.1 for an overview of the currency converter.

USING THE TUSCANY SCA CORBA BINDING ON A SERVICE

In order to expose a service as a CORBA service, you'll need to add the Tuscany CORBA binding to the `<service>` element in the composite XML file. The code for this can be found in the contributions/currency-corba directory of the SCA Tours application and is shown here.

Listing 7.12 The currency converter component with an SCA CORBA binding

```
<composite ...
    xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
    name="currencyconverter">

    <component name="CurrencyConverter">
        <implementation.java class="com.tuscanyscatours.
        ➥ currencyconverter.impl.CurrencyConverterImpl" />
        <service name="CurrencyConverter">
            <interface.java interface="com.tuscanyscatours.
            ➥ currencyconverter.CurrencyConverter" />
```

```
            <tuscany:binding.corba
                host="localhost" port="5080"
                name="CurrencyConverterCORBAService"/>
        </service>
    </component>
</composite>
```

Listing 7.12 shows the currency converter composite XML that includes the SCA CORBA binding 1. Because the CORBA binding is a Tuscany binding, you'll need to import the Tuscany XML namespace and prefix the `binding.corba` with `tuscany`. The host and the port numbers must be specified to configure the host and port numbers for the CORBA service. In this example, you used `localhost` as the host and port `5080`. The final attribute is the `name` attribute, which is used to specify the name that should be associated with the CORBA service and in this case is `CurrencyConverterCORBAService`.

RUNNING THE CURRENCY CONVERTER CORBA SERVICE LAUNCHER

You can use the currency converter CORBA launcher in the launchers/currency-converter-corba directory of the SCA Tours application to test the currency converter using the CORBA binding. When it's run, it produces the same output as previous runs of the currency converter. But this time the currency converter is exposed as a CORBA service. How can you verify that it does? The best way is to write a CORBA client that will access it. Let's do that now.

ACCESSING THE CURRENCY CONVERTER SCA CORBA SERVICE USING A JAVA CLIENT

For any CORBA service, there's an associated Interface Definition Language (IDL) file that describes the CORBA service and the methods it exposes. This is similar in concept to WSDL, which is used to describe web services.

Typically when creating a CORBA client, you'd start from the IDL file for the CORBA service that you want to access. How do you obtain the IDL file for the currency converter CORBA service? Unfortunately, unlike with web services, there's no easy way to download the IDL file. It's possible to convert a Java interface to an IDL file using a Java-to-IDL compiler, but to keep things simple, you'll create the IDL file by hand. The IDL file for the currency converter CORBA service can be found in the clients/currency-converter-corba directory of the SCA Tours application and is shown in the following listing.

Listing 7.13 Currency converter CORBA service IDL file

```
module com {
  module tuscanyscatours {
    module currencyconverter {

      interface CurrencyConverter {

        double getExchangeRate(in string fromCurrencyCode,
          in string toCurrencyCode);

        double convert(in string fromCurrencyCode,
          in string toCurrencyCode, in double amount);
      };
    };
```

```
  };
};
```

The IDL document begins with the module definitions. These are functionally equivalent to Java package definitions. This IDL defines a module called com.tuscanyscatours.currencyconverter. In this module structure is the interface declaration of the `CurrencyConverter` CORBA service. This interface declaration is used to describe the methods on the CORBA service, in this case, the `getExchangeRate()` and `convert()` methods.

Now that you have the IDL for the currency converter CORBA service, you'll need to convert it into a form that can be used in a Java application. To do this, you'll use an IDL-to-Java compiler, which converts an IDL file into Java source code that can be used to access the CORBA service described in the IDL. The JDK comes with an IDL-to-Java compiler called idlj, and the following command converts our currency converter IDL into Java code (from the src/main/resources directory).

```
idlj -fclient -td idlj_output currency-converter.idl
```

This `idlj` command reads the currency-converter.idl file and converts it to Java source code in the idlj_output directory. The `-fclient` option is used to tell idlj to generate the files required for a CORBA client.

You can now use the Java classes generated by idlj to write an application that will use the currency converter CORBA service. The following code shows such a CORBA client.

Listing 7.14 Fragment of the currency converter CORBA client

```
public static void main(String[] args) throws Exception {

  String[] orbArgs = {"-ORBInitialPort", "5080"};
  ORB orb = ORB.init( orbArgs, null );

  String ior = "corbaname::localhost:5080"
     + "#CurrencyConverterCORBAService";

  Object obj = orb.string_to_object(ior);
  CurrencyConverter converter
      = CurrencyConverterHelper.narrow(obj);

  System.out.println("USD -> GBP = "
       + converter.getExchangeRate("USD", "GBP"));
  System.out.println("100 USD = "
       + converter.convert("USD", "GBP", 100.0) + "GBP");
}
```

1 Look up currency converter

2 Test currency converter

The first thing that the currency converter CORBA client does is to initialize the CORBA ORB (Object Request Broker) with the port number 5080. This is the same port number that we specified in the CORBA binding. Next, the code constructs a CORBA Interoperable Object Reference (IOR) and uses it to look up the currency converter CORBA service 1. The IOR that we'll use consists of four parts:

- `corbaname`—The type of the IOR that we'll be using
- `localhost`—The host where the CORBA service is running

- `5080`—The port where the CORBA service is running
- `CurrencyConverterCORBAService`—The name of the CORBA service

Once the currency converter CORBA service has been looked up, it's then used to do some test currency conversions ❷.

CORBA Interoperable Object Reference (IOR)

A CORBA Interoperable Object Reference is a reference that defines the address of a CORBA service. A CORBA client can use an IOR to look up a CORBA service so that it can invoke operations on it.

A CORBA IOR can be used to look up CORBA services across different programming languages (for example, a Java client can look up a COBOL server) and across different CORBA vendor implementations. This is why it's an interoperable object reference.

Running your currency converter CORBA client will produce the following output to the console:

```
USD-> GBP = 0.5
100 USD = 50.0 GBP
```

This output shows that you've been able to invoke the currency component service exposed using the SCA CORBA binding. Let's now look at using the CORBA binding with references.

7.5.2 Accessing a CORBA service using the SCA CORBA binding

Suppose that you wanted to access an existing CORBA service from an SCA application. How would you do this? The answer is to use the SCA CORBA binding on a reference. To illustrate this, you'll again use the Notification service and this time add a Tuscany CORBA binding to the SMS gateway reference. See section 7.2.2 for an overview of the Notification service.

CONNECTING AN SCA REFERENCE TO THE SMS GATEWAY CORBA SERVICE

To connect the Notification service to the SMS gateway CORBA service, we'll add a CORBA binding to the reference.

Listing 7.15 shows the XML composite file for the `notification` contribution with the CORBA binding. The code for this can be found in the contributions/notification-corba directory of the SCA Tours application.

Listing 7.15 Notification composite XML with SMS gateway reference CORBA binding

```
<composite ...
  xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
  name="notification">

  <component name="Notification">
    ...
```

```
    <service name="Notification">
        ...
    </service>

    <reference name="smsGateway">
      <interface.java
        interface="com.tuscanyscatours.smsgateway.SMSGateway"/>
      <tuscany:binding.corba
        host="localhost" port="5080"
        name="SMSGatewayCORBAService"/>
    </reference>
  </component>
</composite>
```

The Notification component has a reference to the SMS gateway. This reference is declared as having a CORBA binding (1) that refers to a CORBA service running on host `localhost` on port `5080` called `SMSGatewayCORBAService`.

RUNNING THE SMS GATEWAY CORBA SERVICE

Before you can start your SMS gateway CORBA service, you'll need to ensure that the CORBA Name Service is running so the SMS gateway can register its CORBA services. To simplify starting the SMS gateway CORBA service, the launcher in the services/smsgateway-corba directory of the SCA Tours application will start both the JDK CORBA Name Service and the SMS gateway CORBA service.

> **CORBA Name Service**
>
> The CORBA Name Service is a standard CORBA service that's used as a registrar for discovering CORBA services based on a service name.
>
> When a CORBA service starts, it can register itself with the CORBA Name Service using a name. A client can then use the CORBA Name Service to look up the CORBA service by using the name that the CORBA service registered with the CORBA Name Service.

When it's run, it displays the following output:

```
Starting transient name server process (port=5080)
tnameserv: Initial Naming Context:
Initial Naming Context:
IOR:000000000000002b49444c3a6f6d67......
TransientNameServer: setting port for initial object references to: 5080
Ready.
Publishing SMS Gateway Service as a CORBA service
CORBA server running - waiting for requests
```

At this point, the CORBA Name Service has started and the SMS gateway CORBA service is running and waiting for requests to process.

RUNNING THE NOTIFICATION SERVICE CONNECTING TO THE SMS GATEWAY CORBA SERVICE

Finally, let's put all the pieces together and get the Notification service to invoke the SMS gateway CORBA service. You'll use the Notification service CORBA launcher from

the launchers/notification-corba directory of the SCA Tours application. When it's run, you'll see the following output on the console:

```
Quick notification test
Sending SMS to +44(0)7700900812 for accountID 1234
Node started - Press enter to shutdown.
```

In addition, the following output will be displayed on the console that started the SMS gateway CORBA service:

```
Sending SMS message
From: +44(0)2079460723
To: +44(0)7700900812
Message: Holiday payment taken. Payment of £102.37 accepted...
```

As you can see from this output, you've been able to successfully invoke the SMS gateway using the SCA CORBA binding from an SCA application.

7.5.3 Configuration options for the SCA CORBA binding

In this section, we'll briefly explore the configuration options for the SCA CORBA binding. The schema for the CORBA binding is shown in here.

Listing 7.16 Schema for the SCA CORBA binding

```
<binding.corba
        name="NCName"
        host="anyURI"
        port="int"
        uri="anyURI"
        requires="sca:listOfQNames"
        policySets="sca:listOfQNames">
<binding.corba/>
```

The SCA CORBA binding has the options shown in table 7.2.

Table 7.2 Options for the SCA CORBA binding

Option	Optional	Description
`name`	Yes	The name for this CORBA binding.
`host`	Yes	The name of the host on which the CORBA service is running.
`port`	Yes	The port number on which the CORBA service is running.
`uri`	Yes	The URI of the CORBA service.
`requires`	Yes	Any required intents for this binding. Not currently supported.
`policySets`	Yes	Any policy sets for this binding. Not currently supported.

You can specify the location of the CORBA service in one of two main ways. It can be specified using the `uri` attribute of `<binding.corba>`, as shown here:

```
<service name="CurrencyConverter">
    <interface.java interface=
        "com.tuscanyscatours.currencyconverter.CurrencyConverter" />
    <tuscany:binding.corba
        uri="corbaname::localhost:5080/NameService#
        ➥ CurrencyConverterCORBAService"/>
</service>
```

Or it can be specified using the `host`, `port`, and `name` attributes of `<binding.corba>`, as follows:

```
<service name="CurrencyConverter">
    <interface.java interface=
        "com.tuscanyscatours.currencyconverter.CurrencyConverter" />
    <tuscany:binding.corba
        host="localhost" port="5080"
        name="CurrencyConverterCORBAService"/>
</service>
```

These two `<binding.corba>` definitions can be used interchangeably because they both specify the same location for the CORBA service.

At this point, you've been able to use the SCA CORBA binding on services and references, and you also understand the configuration options available for it. Let's move on and have a look at another binding, the SCA RMI binding.

7.6 Connecting component services with RMI

In this section we'll look at how the SCA RMI binding, `binding.rmi`, can be used to expose services over RMI and how you can access RMI services in SCA. Feel free to read this section later if you're currently not planning to use RMI in your SCA applications.

What is RMI?

The Java Remote Method Invocation (RMI) API allows Java applications to invoke operations on objects in another JVM that have been exposed as RMI services. This is often referred to as a Remote Procedure Call (RPC).

The typical usage scenario is where a Java server application exposes one or more objects as remote objects that can be accessed by RMI clients. Once the server has exposed these remote objects, it typically waits for clients to invoke them. A Java client looks up these remote objects in the RMI registry and invokes operations on them. The server receives the requests and performs the required business logic, returning any results to the client.

RMI is provided as a core part of the Java language.

The Tuscany RMI binding doesn't have an associated SCA specification. Tuscany implements the RMI binding using the extensibility features from the SCA specifications. As a result, this binding is Tuscany-specific and may not be available on other SCA implementations.

Outside of Tuscany and SCA, RMI is specific to the Java programming language and is provided as part of the JDK you'll use to write Java programs. To write Java clients and services that exploit RMI, you'll use the Java API for declaring remote interfaces, exposing them for use, and for finding them in the RMI registry on the client side.

With Tuscany, the component developer doesn't use these APIs but adds the `<binding.rmi>` element to a component's services and references as appropriate. This process is the same as for any SCA binding, and `binding.rmi` isn't restricted to components using the Java implementation.

Let's start by looking at using `binding.rmi` on a component service.

7.6.1 Exposing an SCA service as an RMI service

Let's explore how the currency converter component can be exposed as an RMI service using the RMI binding.

USING THE TUSCANY SCA RMI BINDING ON A SERVICE

To expose the currency converter as an RMI service, you'll need to add the Tuscany RMI binding to the `<service>` element in the composite XML file. The code for this can be found in the contributions/currency-rmi directory of the SCA Tours application and is shown next.

Listing 7.17 Currency converter composite with Tuscany SCA RMI binding

```
<composite ...
    xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
    name="currencyconverter">

    <component name="CurrencyConverter">
        <implementation.java class="com.tuscanyscatours.
        ➥ currencyconverter.impl.CurrencyConverterImpl" />
        <service name="CurrencyConverter">
            <interface.java interface="com.tuscanyscatours.
            ➥ currencyconverter.CurrencyConverter" />
            <tuscany:binding.rmi
                host="localhost" port="8099"
                serviceName="CurrencyConverterRMI"/>
        </service>
    </component>
</composite>
```

1 RMI port and host

2 RMI service name

Listing 7.17 shows the modified currency converter composite XML file with the Tuscany RMI binding added. You'll need to import the Tuscany XML namespace because the RMI binding isn't defined by an SCA specification. Because it's part of the RMI binding, you'll need to specify the host and the port details 1 and the service name 2 for the RMI service. These details will be used by any RMI clients to look up the currency converter RMI service.

Let's move on to create an RMI client to check that the service has been exposed as an RMI service.

ACCESSING THE CURRENCY CONVERTER SCA RMI SERVICE USING A JAVA CLIENT

Writing RMI clients is relatively simple and involves looking up the RMI service to get a proxy to the service and then invoking operations on that proxy. The following listing shows the currency converter RMI client that will invoke the currency converter over RMI. The code can be found in the clients/currency-converter-rmi directory of the SCA Tours application.

Listing 7.18 Fragment of the currency converter RMI client

```
public static void main(String[] args) throws Exception {
  Registry registry = LocateRegistry.getRegistry(
      "localhost", 8099);

  String name = "CurrencyConverterRMI";
  CurrencyConverter converter
      = (CurrencyConverter) registry.lookup(name);

  System.out.println("USD -> GBP = "
      + converter.getExchangeRate("USD", "GBP"));
  System.out.println("100 USD = "
      + converter.convert("USD", "GBP", 100.0) + "GBP");
}
```

(1) Connect to RMI registry

(2) Look up currency converter RMI service

To look up an RMI service, the client needs to use the RMI registry where the RMI services are registered. To obtain the RMI registry, the client calls the `getRegistry()` method on `LocateRegistry` and specifies the same hostname and port (1) as specified in the currency converter composite XML file.

Each service in the RMI registry is registered against a unique name. This name is required to look up an RMI service. In our case, the RMI binding on the currency converter component used the service name CurrencyConverterRMI (2).

Running this currency converter RMI client will produce the following output. (Note: make sure the currency converter RMI SCA service launcher, launchers/currency-converter-rmi, is running first.)

```
USD -> GBP = 0.5
100 USD = 50.0GBP
```

This output shows that you've successfully invoked the currency converter service over RMI. Let's now look at using the RMI binding on references.

7.6.2 Accessing an RMI service using the SCA RMI binding

By using the RMI binding on a reference, it's possible to invoke an existing RMI service from an SCA application. In this section, we'll use the Notification service from section 7.2.2 again but this time add an RMI binding to the SMS gateway service.

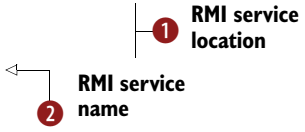
CONNECTING AN SCA REFERENCE TO THE SMS GATEWAY RMI SERVICE

Let's update the Notification service composite XML file so that it can invoke the SMS gateway over RMI. To do this, you'll need to add the `<binding.rmi>` element to the

reference. The code for this can be found in the contributions/notification-rmi directory of the SCA Tours application and is shown here.

Listing 7.19 The SMS gateway reference with SCA RMI binding

```
<composite ...
    xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
    name="notification">

    <component name="Notification">
        ...
        <service name="Notification">
            ...
        </service>

        <reference name="smsGateway">
            <interface.java
                interface="com.tuscanyscatours.smsgateway.SMSGateway"/>
            <tuscany:binding.rmi
                host="localhost" port="8099"
                serviceName="SMSGatewayRMI"/>
        </reference>
    </component>
</composite>
```



The reference to the SMS gateway in the Notification component now uses an RMI binding. Because the RMI binding is in the Tuscany XML namespace, you'll need to import it. The RMI binding has attributes that define the host and port (1) and the service name (2) that should be used to connect to the SMS gateway RMI service.

RUNNING THE SMS GATEWAY RMI SERVICE

Before you can use the SMS gateway RMI service in your SCA application, you'll need to ensure that it's running using the launcher in the services/smsgateway-rmi directory of the SCA Tours application. When started, the SMS gateway RMI service will display the following output:

```
Publishing SMS Gateway Service as a RMI service
RMI server running - waiting for requests
Press enter to shutdown.
```

The SMS gateway RMI service has been started and is listening on its RMI interface for requests to send SMS messages.

RUNNING THE NOTIFICATION SERVICE CONNECTING TO THE SMS GATEWAY RMI SERVICE

Let's see whether your Notification service application with the RMI binding can invoke the SMS gateway RMI service. To do this, we'll run the Notification service RMI using the launcher in the launchers/notification-rmi directory of the SCA Tours application, which will produce the following console output:

```
Quick notification test
Sending SMS to +44(0)7700900812 for accountID 1234
Node started - Press enter to shutdown.
```

In addition, the following output will be displayed on the console where you started the SMS gateway RMI service:

```
Sending SMS message
From: +44(0)2079460723
To: +44(0)7700900812
Message: Holiday payment taken. Payment of £102.37 accepted...
```

As you can see from the output, you've been able to successfully invoke the SMS gateway using the SCA RMI binding from an application.

7.6.3 Configuration options for the SCA RMI binding

In this section, we'll briefly explore the configuration options for the SCA RMI binding. The schema for the RMI binding is shown in the following listing.

Listing 7.20 Schema for the SCA RMI binding

```
<binding.rmi
        name="NCName"
        uri="anyURI"
        host="anyURI"
        port="int"
        serviceName="anyURI"
        requires="sca:listOfQNames"
        policySets="sca:listOfQNames">
<binding.rmi/>
```

The RMI binding has the following options, as shown in table 7.3.

Table 7.3 Options for the SCA RMI binding

Option	Optional	Description
`name`	Yes	The name for this binding.
`uri`	Yes	The URI of the RMI service.
`host`	Yes	The name of the host on which the RMI service is running.
`port`	Yes	The port number on which the RMI service is running.
`serviceName`	Yes	The name of the RMI service to connect to.
`requires`	Yes	Any required intents for this binding. Not currently supported in Tuscany.
`policySets`	Yes	Any policy sets for this binding. Not currently supported in Tuscany.

The location of the RMI service can be specified in one of two main ways. It can be specified using the `uri` attribute of `<binding.rmi>`, as shown here:

```
<service name="CurrencyConverter">
    <interface.java interface=
        "com.tuscanyscatours.currencyconverter.CurrencyConverter" />
```

```
    <tuscany:binding.rmi
        uri="rmi://localhost:8099/CurrencyConverterRMI"/>
</service>
```

Or it can be specified using the `host`, `port`, and `serviceName` attributes of `<binding.rmi>` as follows:

```
<service name="CurrencyConverter">
    <interface.java interface=
        "com.tuscanyscatours.currencyconverter.CurrencyConverter" />
    <tuscany:binding.rmi
        host="localhost" port="8099"
        serviceName="CurrencyConverterRMI"/>
</service>
```

These two `<binding.rmi>` definitions can be used interchangeably because they both specify the same location for the RMI service.

You now know how to use the RMI binding to expose services as RMI services and how to invoke RMI services using references. Let's look at the SCA JMS binding next.

7.7 Connecting component services with JMS

So far we've concentrated on technologies where a request is sent and a response is received. SCA also defines bindings for message-oriented technologies like JMS, where the protocol's interaction pattern is designed to be asynchronous. Feel free to read this section later if you're currently not planning to use JMS in your SCA applications.

> **What is JMS?**
>
> The Java Message Service (JMS) API allows Java applications to send and receive messages. It allows applications to communicate using asynchronous messages containing business events or data.

The beauty of using SCA is that the JMS binding, `binding.jms`, is applied to composite applications in the same way as any other binding. Without SCA and Tuscany, the Java language-specific JMS API is used directly from your Java programs to place messages onto queues and retrieve messages from queues. When using Tuscany, it's not necessary to use the JMS API. The addition of the `<binding.jms>` element to a component's services and/or references is sufficient to enable asynchronous, message-based communication.

The JMS API provides numerous configuration options, and, as you shall see later in this section, `binding.jms` offers many of these options. But let's start by looking at using `binding.jms` on a component service.

7.7.1 Exposing an SCA service using JMS

In this section we'll show how the currency converter component from section 7.2.1 can be exposed as a JMS service using the SCA JMS binding and access it using a JMS client.

This will involve adding the `<binding.jms>` element to the `service` element of the composite XML file.

USING THE SCA JMS BINDING ON A SERVICE

Let's see how the JMS binding can be used to make services of the currency converter available to others. To do this, we'll add the JMS binding to the `<service>` element of the currency converter composite XML file. The code for this can be found in the contributions/currency-jms directory of the SCA Tours application and is shown here.

Listing 7.21 Currency converter component with JMS binding added

```
<composite ...
  name="currencyconverter">
  <component name="CurrencyConverter">
    <implementation.java
      class="com.tuscanyscatours.currencyconverter.impl.
      ➥ CurrencyConverterImpl" />
    <service name="CurrencyConverter">
      <interface.java interface=
        "com.tuscanyscatours.currencyconverter.CurrencyConverter" />
      <binding.jms
        initialContextFactory
          ="org.apache.activemq.jndi.ActiveMQInitialContextFactory"
        jndiURL="tcp://localhost:61619">
        <destination name="RequestQueue" create="ifnotexist"/>
        <response>
          <destination name="ResponseQueue" create="ifnotexist"/>
        </response>
      </binding.jms>
    </service>
  </component>
</composite>
```

1 JMS binding

Listing 7.21 shows how the currency converter component would look with the addition of the JMS binding 1. Unlike many of the other SCA service bindings, you'll need to specify some configuration options to configure the JMS binding. You'll need to specify the class name of the JMS initial context factory that's used to look up the JMS message broker and the URL of the JNDI service that holds the JMS message broker objects. Tuscany uses Apache ActiveMQ (http://activemq.apache.org/) as the JMS message broker. Finally, you'll need to configure the request queue that's used to receive JMS messages and the response queue that's used to return response messages to the caller.

ACCESSING THE CURRENCY CONVERTER SCA SERVICE USING A JMS CLIENT

You can send messages to the currency converter service through a JMS client that looks like the following listing. The code for the JMS client can be found in the clients/currency-converter-jms directory of the SCA Tours application.

Listing 7.22 Fragment of the currency converter JMS client

```
public static void main(String[] args) throws Exception {
  startActiveMQSession();

  OMElement request = omFactory.createOMElement(
      "convert", "http://goodvaluetrips.com/", "ns2");
  request.addChild(createArg(0, "USD"));
  request.addChild(createArg(1, "GBP"));
  request.addChild(createArg(2, "100.0"));
  TextMessage message
      = activeMQSession.createTextMessage("convert");
  message.setStringProperty(
      "scaOperationName", "convert");
  message.setJMSReplyTo(responseDest);
  message.setText(request.toString());

  activeMQProducer.send(message);

  TextMessage response =
    (TextMessage) consumer.receive();

  StAXOMBuilder builder = new StAXOMBuilder(
      new ByteArrayInputStream(response.getText().getBytes()));
  OMText returnElement = (OMText)
      builder.getDocumentElement().getFirstOMChild();
  String returnValue = returnElement.getText();
  System.out.println("100 USD = " + returnValue + "GBP");

  stopActiveMQSession ();
}
```

1 Create JMS message

2 Send JMS message

3 Receive JMS response

In listing 7.22, you first started the ActiveMQ session by calling the `startActiveMQSession` method (not shown) that connects to the JMS message broker and creates the request and response queues using the same queue names that you specified on the JMS binding on the currency converter.

The next step would be to create the JMS message 1 body that contains the parameters for the `convert` request. When the default wire format is used for the JMS binding, XML is used to encode the request details in the message body of the JMS message. Here you use Apache Axis Object Model (AXIOM) (http://ws.apache.org/commons/axiom/) for mapping objects to XML. You'll first need to create an `OMElement` to represent the `convert` request, with the XML namespace matching the namespace of the currency converter. Next, you'll need to add the parameters for the `convert` request by adding a child `OMElement` (named `arg0` `arg1`, `arg2`, ...) to the request for each parameter. To save duplicating the same code for each parameter, you'll use the `createArg` helper method to add the parameters.

Now that you have the JMS message body, you'll need to create the JMS message. SCA can use JMS text messages to send requests, so you'll create a JMS text message that specifies the operation that you want to invoke (in this case `convert`) and attaches the message payload containing the parameters as XML from the `OMElement` request you created earlier.

What is AXIOM?

AXIOM is an Apache project that provides an XML Infoset-compliant object model that can be used to convert Java objects into XML and back. It provides an API that allows primitive and complex object types to be converted and expressed as an XML Infoset. It can also perform the reverse conversion and convert an XML Infoset back into primitive and complex object types.

Your `convert` request has now been packaged as a JMS message and is ready to be sent to the currency converter service for processing. This is done by using the `producer` to send the JMS message ❷.

At this point, you'll wait for the JMS message broker to deliver your JMS message to the Email Gateway component and for it to reply with another JMS message. You'll receive the JMS reply using the message consumer ❸. The response will be in the form of more XML, so you'll need to parse the returned XML to retrieve the return value before you can display it on the console.

Before we run the currency converter JMS client, we'll need to start the currency converter JMS server by using the launcher in the launchers/currency-converter-jms directory. When it is run, it will produce the following output:

```
ActiveMQ null JMS Message Broker (localhost) is starting
Starting node: currency-converter-jms.composite
JMS service 'CurrencyConverter' listening on destination RequestQueue
Node started - Press enter to shutdown.
```

The JMS client can be found in the clients/currency-converter-jms directory. When it is run, it produces the following output:

```
100 USD = 50.0GBP
```

It doesn't look like much, but the JMS client did send a JMS message to the currency converter and received in reply another JMS message. You can verify this by stopping the JMS message broker and running the JMS currency converter client again. This time you should get a `JMSException`.

Now that you've mastered the JMS binding on a service, let's move on to using the JMS binding on a reference.

7.7.2 Accessing a JMS service using the SCA JMS service binding

By using the SCA JMS binding on a reference, an SCA application can interact with JMS services using JMS messages. In this section, we'll use the Notification service and update it so it can make use of an SMS gateway that exposes a JMS interface. See section 7.2.2 for an overview of the Notification service and SGS gateway.

CONNECTING AN SCA REFERENCE TO THE SMS GATEWAY JMS SERVICE

Once again, let's update the Notification service so that this time it will be able to invoke the SMS gateway using JMS. To do this, you'll add the `<binding.jms>` element

to the reference. The code for this can be found in the contributions/notification-jms directory of the SCA Tours application and is shown next.

Listing 7.23 SMS gateway reference with SCA JMS binding

```
<composite name="notification" ...>
  <component name="Notification">
    <service name="Notification">
      ...
    </service>
    <reference name="smsGateway">
      <interface.java
        interface="com.tuscanyscatours.smsgateway.SMSGateway"/>
      <binding.jms
        initialContextFactory
          ="org.apache.activemq.jndi.ActiveMQInitialContextFactory"
        jndiURL="tcp://localhost:61619">
        <destination name="SMSRequestQueue" create="ifnotexist"/>
        <response>
          <destination name="SMSResponseQueue" create="ifnotexist"/>
        </response>
      </binding.jms>
    </reference>
  </component>
</composite>
```

Listing 7.23 shows the currency converter composite with the addition of a JMS binding. You have to specify the JMS initial context factory and the URL of the JMS message broker. Finally, the request and response JMS queues are configured. Notice how easy this is compared to the complexities of using JMS in the non-SCA client code shown previously.

RUNNING THE NOTIFICATION SERVICE CONNECTING TO THE SMS GATEWAY JMS SERVICE

Before running the Notification service, you'll need to ensure that the JMS message broker and the SMS gateway JMS service are running using the launcher in the services/smsgateway-jms directory of the SCA Tours application. Once it's running, you can run the Notification service using the launcher in the launchers/notification-jms directory of the SCA Tours application. It will produce the following output on the console:

```
Quick notification test
Sending SMS to +44(0)7700900812 for accountID 1234
Node started - Press enter to shutdown.
```

The following output will also be displayed on the console where the SMS gateway was started:

```
Sending SMS message
From: +44(0)2079460723
To: +44(0)7700900812
Message: Holiday payment taken. Payment of £102.37 accepted...
```

As you can see from these two outputs, the Notification service has looked up the mobile phone number for account ID 1234 and used the JMS interface of the SMS gateway to send the user an SMS.

We've already used some of the configuration options that are available with the JMS binding. Let's move on and look at them in a little more detail.

7.7.3 Configuration options for the SCA JMS binding

This section outlines the configuration options that are available for the JMS binding. We've not included the full schema because the JMS binding has many options, and to describe them all would require a whole chapter in its own right. See the OSOA SCA JMS binding specification for a full description of all the available JMS binding options. It can be downloaded from http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

The key elements of the schema for the JMS binding are shown here.

Listing 7.24 Schema for the SCA JSM binding

```
<binding.jms
        name="NCName"
        uri="anyURI"
        initialContextFactory="anyURI"
        jndiURL="anyURI"
        ...
        >
    <destination name="xs:anyURI" type="string" create="string">
        ...
    </destination>
    ...
    <response>
        <destination name="xs:anyURI" type="string" create="string">
            ...
        </destination>
        ...
    </response>
    ...
    requires="sca:listOfQNames"
    policySets="sca:listOfQNames">
   </binding.jms>
```

The JMS binding has the following configuration options, as shown in table 7.4.

Table 7.4 Configuration options for the SCA JMS binding

Option	Optional	Description
`name`	Yes	The name for this binding.
`uri`	Yes	The URI that can be used to configure JMS.
`initialContextFactory`	Yes	The initial context factory.
`jndiURL`	Yes	The URL of the JNDI provider.
`destination`	Yes	The destination to which the JMS binding should send messages.
`destination/type attribute`	Yes	Whether the destination is a `queue` or a `topic`, with `queue` being the default.

Table 7.4 Configuration options for the SCA JMS binding *(continued)*

Option	Optional	Description
`destination/create attribute`	Yes	Whether the destination should be created or not. Valid values are `always`, `ifnotexist`, and `never`, with `ifnotexist` being the default.
`Response`	Yes	The location from which the JMS binding should expect responses to its requests.
`response/type attribute`	Yes	Whether the response is a `queue` or a `topic`, with `queue` being the default.
`response/create attribute`	Yes	Whether the response should be created or not. Valid values are `always`, `ifnotexist`, and `never`, with `ifnotexist` being the default.
`Requires`	Yes	Any required intents for this binding.
`policySets`	Yes	Any policy sets for this binding.

One of the more useful attributes of the `destination` and `response` elements is the `create` attribute. This is used to specify whether the JMS queues should be created or not when the SCA application is run. The default value is `ifnotexist`, which means that the JMS queue will be created if it doesn't exist. It can be set to `never`, which means that the JMS queue won't be created but must already exist for the application to run. It could be set to `always`, which means that the JMS queue will be created and must not exist for the application to run.

Now that you understand how a message-oriented binding can be used in SCA, let's move on to our next SCA binding, the EJB binding.

7.8 Connecting to EJBs

In this section, we'll examine how the SCA EJB binding, `binding.ejb`, can be used to invoke EJBs running on a Java Enterprise Edition server (such as Apache Geronimo) from an SCA application. Feel free to read this section later if you're currently not planning to use EJBs in your SCA applications.

What are Enterprise JavaBeans?

Enterprise JavaBeans (EJBs) are a server-side technology that provides a component-based model for implementing business logic. EJBs run in an EJB server that provides a secure and scalable runtime, including infrastructure services such as transactions, data persistence, and so on that are designed to simplify the task of writing business logic.

Outside of SCA, EJB is a Java language–specific technology. Communicating with an EJB involves a protocol based on the Java language–specific RMI protocol we discussed earlier in this chapter. But when using Tuscany and SCA, you can add a `<binding.ejb>`

element to a component reference in the same way that any other binding can be added. In this way you're able to communicate with EJBs without having to worry about whether it has an EJB home interface (for EJB 2) or not (for EJB 3) or how to narrow remote object references. Let's look at how `binding.ejb` is used with services and references.

7.8.1 Exposing an SCA service as an EJB

At the time of writing this book, the EJB binding in Tuscany doesn't support exposing a service as an EJB. If you need this feature, check the Tuscany website for the latest development information.

7.8.2 Accessing an EJB using the SCA EJB binding

By attaching an SCA EJB binding to a reference, it's possible to invoke an existing EJB running in a Java Enterprise Edition container from an SCA application. Let's use the Notification service and add an EJB binding to the SMS gateway reference to access an EJB. See section 7.2.2 for an overview of the Notification service.

The EJB specification defines several types of EJBs, with stateless session beans, stateful session beans, and entity beans being the most common types. The EJB binding specification defines how the binding can be used to connect to stateless and stateful EJB session beans. It doesn't support invoking EJB entity or message-driven beans.

CONNECTING AN SCA REFERENCE TO THE SMS GATEWAY EJB

To connect the Notification service to the SMS gateway EJB service, you need to add an SCA EJB binding to the reference in the Notification component. The code for this can be found in the contributions/notification-ejb directory of the SCA Tours application and is shown in this listing.

Listing 7.25 Notification composite XML with SMS gateway reference EJB binding

```
<composite ...
  name="notification">
  <component name="Notification">
    ...
    <service name="Notification">
        ...
    </service>

    <reference name="smsGateway">
      <interface.java
        interface="com.tuscanyscatours.smsgateway.SMSGateway"/>
      <binding.ejb
        uri="SMSGatewayImplRemote"/>
    </reference>
  </component>
</composite>
```

1 EJB binding

2 EJB name

In listing 7.25, the SMS gateway reference of the Notification component has been updated to include the EJB binding by adding `<binding.ejb>` 1. The `uri` attribute 2 of the EJB binding is used to specify the location of the EJB. Because in this example the

EJB will be run with Apache OpenEJB, in this case we're using the name of the remote interface of the SMS gateway EJB. If you're using a different EJB server, then the format of the `uri` attribute may be different. See your EJB server documentation for more detail.

RUNNING THE SMS GATEWAY EJB SERVICE

The SMS gateway EJB is a stateless session bean that will simulate sending an SMS message. In order for the SMS gateway EJB service to be run, you need to use a Java Enterprise Edition container and deploy your SMS gateway EJB bean to it. In order to keep the EJB handling simple, you'll use the Apache OpenEJB Java Enterprise Edition container that can be started in standalone mode. To start the OpenEJB server and deploy your SMS gateway EJB, you can run the launcher in the services/smsgateway-ejb directory of the SCA Tours application. When it's run, it displays some standard OpenEJB startup messages and then displays the following output:

```
Publishing SMS Gateway Service as an EJB service
EJB server running - waiting for requests
```

At this point, the OpenEJB server is running, and the SMS gateway EJB has been deployed and is waiting for requests to process.

RUNNING THE NOTIFICATION SERVICE CONNECTING TO THE SMS GATEWAY EJB SERVICE

Let's use the EJB launcher from the launchers/notification-ejb directory of the SCA Tours application to run the Notification service. This will connect to the SMS gateway running as an EJB, and you'll see the following on the console:

```
Quick notification test
Sending SMS to +44(0)7700900812 for accountID 1234
Node started - Press enter to shutdown.
```

In addition, the following output will be displayed on the console that started the OpenEJB server containing the SMS gateway EJB:

```
Sending SMS message
From: +44(0)2079460723
To: +44(0)7700900812
Message: Holiday payment taken. Payment of £102.37 accepted...
```

As you can see from this output, you've successfully invoked the SMS gateway using the SCA EJB binding from an SCA application.

7.8.3 Configuration options for the SCA EJB binding

The EJB binding has several options that can be used to configure it. These are shown in the schema in listing 7.26.

Listing 7.26 Schema for the EJB binding

```
<binding.ejb
        name="NCName"
        uri="anyURI"
        homeInterface="NCName"
```

```
        ejb-link-name="NCName"
        session-type="stateful or stateless"
        ejb-version="EJB2 or EJB3"
        requires="listOfQNames"
        policySets="listOfQNames">
<binding.ejb/>
```

The EJB binding has the following options, as shown in table 7.5.

Table 7.5 Options for the SCA EJB binding

Option	Optional	Description
`name`	Yes	The name for this binding.
`uri`	Yes	For services, the URI where the EJB should be published. For references, the URI of the EJB to connect to.
`homeInterface`	Yes	The EJB2 home interface. Not used for EJB3.
`ejb-link-name`	Yes	Used to link to an EJB in the same Java Enterprise Edition EAR.
`session-type`	Yes	Used to specify whether the session bean is stateless or stateful. If not specified, it will be deduced from the EJB interface.
`ejb-version`	Yes	EJB2 or EJB3.
`requires`	Yes	Any required intents for this binding.
`policySets`	Yes	Any policy sets for this binding.

See the OSOA SCA EJB session bean binding specification for a more detailed description of all the options for the EJB binding. It can be downloaded from http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

This concludes our tour of some of the SCA bindings supported by Tuscany. Let's move on and wrap up what you've learned in this chapter.

7.9 Summary

In this chapter, you've seen how, by using SCA bindings, you can expose an SCA service and access external services using a variety of communication protocols. Whether you want to interact with web services, CORBA services, RMI services, JMS, or EJB, there's a binding that will help you.

The key point to take away from this chapter is that you've been able to add one or more SCA bindings without changing the component implementation code. The binding approach provides a clean separation between the application logic contained with the component implementation and the communication protocols used to wire them together.

At this point, it's a good time to shift our focus toward the exciting world of web clients and Web 2.0 to see what Tuscany and SCA offer. You'll find a few more web-related bindings lurking among the information in the next chapter.

8 Web clients and Web 2.0

This chapter covers

- Using SCA with servlets and JSPs
- Creating interactive web interfaces using SCA and JavaScript
- Integrating Atom and RSS feeds into SCA applications

In today's computing world, everything seems to be web-enabled and available over the internet, for example, banking, food shopping, business applications, your utility bills, and so on.

There are many ways to make applications web-enabled. These include the more traditional approaches of using a web-enabling framework, for example, the Java Platform, Enterprise Edition, and the less-traditional approaches encompassed by the term *Web 2.0*, such as Ajax, Atom, and RSS feeds, and the like.

In this chapter, we'll look at some of the ways that SCA and Tuscany can help to simplify web-enabling your applications. First, we'll look at how SCA can be used with Java Enterprise Edition by looking at Java Servlets (or servlets for short) and JavaServer Pages (JSPs). The advantage of this approach is that organizations that have already invested time in creating Java Enterprise Edition applications can reuse their existing code in new SCA applications.

We'll then move on to look at how you can use HTML and JavaScript to create interactive browser-based clients that make use of SCA services. The advantage of this approach is that interactive clients can be produced without requiring a full-featured Java Enterprise Edition server because it will run with a much lighter-weight web server. Not all elements within a web application need to be interactive, and some can be static, for example, static HTML pages, images, and logos. We'll also look at how Tuscany can help with this type of static content.

Finally, we'll look at Atom and RSS feeds. We'll show how to produce Atom and RSS feeds using Tuscany. People can subscribe to these feeds and be notified of updates. We'll then look at how to consume Atom and RSS feeds using Tuscany.

There's a lot of information to cover in this chapter, so let's get started and look at using servlets with SCA.

8.1 Servlets as SCA component implementations

In this section, we'll look at how you can create an interactive web application using SCA and servlets. Servlets are part of the Java Enterprise Edition specification and allow developers to create dynamic web content using the Java programming language. We'll examine how a servlet can be used as a component implementation and how it can invoke existing SCA services.

By using a servlet as a component implementation, a developer can create applications that combine existing servlets or Java Enterprise Edition code with new or existing SCA services. This hybrid approach maximizes reuse of existing code and allows SCA to be adopted incrementally.

8.1.1 Creating the currency converter user interface using a servlet

Suppose that as part of TuscanySCATours, you wanted to provide your customers with a web interface that they can use to calculate the conversion value of currencies, for example, convert $100 U.S. into sterling.

The TuscanySCATours application already contains an SCA currency converter contribution that allows values in one currency to be converted to another. All you'll need to do is create a web interface that uses it. In this section, you'll create a servlet that will use the currency converter contribution to provide a web interface that can be used to convert currencies, as shown in figure 8.1.

The currency converter servlet runs within a Java Enterprise Edition container and uses the currency converter SCA component to convert currencies. A web browser can be used to invoke the currency converter servlet over HTTP.

USING TUSCANY IN A SERVLET

A servlet can look up and use an SCA service. How this is achieved depends on the level of support the servlet container provides for SCA. If SCA is fully supported, then the servlet code can use the `@Reference` annotation as it would in any other SCA code. But few servlet containers today contain this level of support for SCA. For these servlet

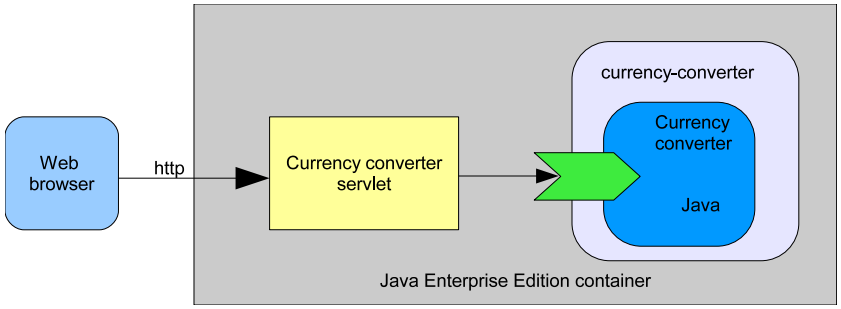


Figure 8.1 Overview of a currency converter servlet connected to the currency converter SCA component

containers, the `@Reference` annotation won't work, and the references need to be set up in the `init()` method of the servlet. Currently, support for SCA is available in Apache Geronimo (using a plug-in) and in IBM WebSphere.

The following listing shows the `CurrencyConverterServlet.java` class from contributions/currency-servlet that makes use of SCA to invoke the currency converter.

Listing 8.1 Currency converter servlet invoking SCA service

```
public class CurrencyConverterServlet extends HttpServlet {

  @Reference
  protected CurrencyConverter currencyConverter;

  public void init(ServletConfig config) {
    if (currencyConverter == null) {
      ComponentContext context =
          (ComponentContext)config.getServletContext()
          .getAttribute("org.osoa.sca.ComponentContext");
      currencyConverter = context.getService(
          CurrencyConverter.class, "currencyConverter");
    }
  }

  protected void service(HttpServletRequest request,
      HttpServletResponse response) throws IOException {
    Writer out = response.getWriter();
    out.write("<html><body><h2>SCA Tours Currency Converter Servlet</h2>");
    out.write("Welcome to the SCA Tours Currency Converter Servlet<p>");
    out.write("<form method=post action=
    ➥ \"CurrencyConverterServlet\">");
    out.write("Enter value in US Dollars");
    out.write("<input type=text name=dollars size=15><p>");
    out.write("<input type=submit>");
    out.write("</form><p>");

    String dollarsStr = request.getParameter("dollars");
    if (dollarsStr != null) {
```

1 **Container-supported reference injection**

2 **Containers not supporting reference injection**

3 **Currency input form**

```
        double dollars = Double.parseDouble(dollarsStr);
        double converted = currencyConverter.convert(
            "USD", "GBP", dollars);
        out.write(dollars + " US Dollars = "
            + converted + " GB Pounds");
    }

    out.write("</body></html>");
    out.flush();
    out.close();
  }
}
```

4 Convert currency using SCA service

The servlet uses the `@Reference` annotation (1) to look up the currency converter service. This style of reference injection will work on servlet containers that support SCA. But because most don't at this time, the servlet has code (2) to look up the references using the SCA component context. The SCA `ComponentContext` is provided as a standard part of the SCA API. The servlet contains a form (3) that's used to enter the amount in U.S. dollars to convert to sterling. The final block of code checks whether an amount in U.S. dollars has been entered, and if so, it converts it into sterling using the currency converter service (4).

CONFIGURING THE SCA CURRENCY CONVERTER SERVICE

Our servlet looks up the currency converter SCA service, but we haven't yet configured it. Let's do that now by creating a META-INF/sca-deployables/web.composite file that contains the composite definition. Tuscany will automatically scan the META-INF/sca-deployables directory for composites. This web.composite file can be found in the contributions/currency-servlet directory of the TuscanySCATours application and is shown here.

Listing 8.2 Currency converter servlet web.composite file

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
    targetNamespace="http://scatours.com"
    name="CurrencyConverterServlet">

    <component name="WebClient">
        <implementation.web web-uri=""/>
        <reference name="currencyConverter"
            target="CurrencyConverter"/>
    </component>

    <component name="CurrencyConverter">
        <implementation.java class=
            "com.tuscanyscatours.currencyconverter.impl.
            ➥ CurrencyConverterImpl"/>
        <service name="CurrencyConverter"/>
    </component>
</composite>
```

1 WebClient currency converter reference

2 Currency converter component

In this composite file the `WebClient` component is defined so that it has a reference to the currency converter service (1). We set `web-uri` to `""` to indicate that the whole of

the WAR is an SCA contribution. The composite file also defines the currency converter component that exposes the currency converter service ❷.

CONFIGURING THE SERVLET FILTER TO ROUTE REQUESTS TO TUSCANY

The final file that we'll need to define for the servlet is the WEB-INF/web.xml file that configures a servlet filter that will direct requests via Tuscany. It can be found in the contributions/currency-servlet directory of the TuscanySCATours application and is shown in the following listing.

Listing 8.3 Servlet filter to direct requests via Tuscany

```
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

    <display-name>SCA Tours Currency Converter Servlet</display-name>

    <filter>
        <filter-name>tuscany</filter-name>
        <filter-class>
            org.apache.tuscany.sca.host.webapp.
            ➥ TuscanyServletFilter
        </filter-class>
    </filter>

    <filter-mapping>
        <filter-name>tuscany</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>CurrencyConverterServlet</servlet-name>
        <servlet-class>com.tuscanyscatours.currencyconverter.
        ➥ servlet.CurrencyConverterServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>CurrencyConverterServlet</servlet-name>
        <url-pattern>/CurrencyConverterServlet</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>CurrencyConverterServlet</welcome-file>
    </welcome-file-list>
</web-app>
```

❶ Tuscany servlet filter

❷ Currency converter servlet

❸ URL mapping

❹ Default web page

The Tuscany servlet filter ❶ is used to start and stop Tuscany when the web application starts and stops. This ensures that the `CurrencyConverterServlet` ❷ can make use of SCA services. The `servlet-mapping` ❸ is used to define the URI to access the servlet. Finally, you'll configure the `CurrencyConverterServlet` to be displayed by default ❹.

At this point, you've defined all the files that you'll need:

- META-INF/sca-deployables/web.composite
- WEB-INF/web.xml

Let's move on to deploying and running the currency converter servlet.

DEPLOYING AND RUNNING THE CURRENCY CONVERTER SERVLET

Before you can run the currency converter servlet, you'll need to create the WAR file and deploy it on a web server that can host servlets. In our sample we'll use the Maven build to produce the WAR file. Please refer to your web server's documentation for information on how to deploy a WAR file. Once the currency converter WAR file is deployed, you can access the currency converter servlet using a web browser: http://localhost:8080/scatours-contribution-currency-servlet/.

You should see the currency converter servlet, as shown in figure 8.2.

Figure 8.2 Currency converter servlet viewed through a web browser

Entering 600 in the field and clicking the Submit Query button will cause the currency converter servlet to convert $600 U.S. into sterling using the SCA currency converter service and display the result, as shown in figure 8.3.

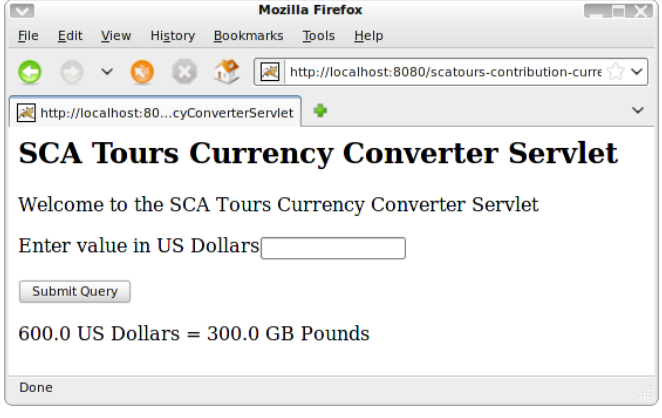


Figure 8.3 Currency converter servlet converted $600 U.S. to sterling

In this section, you've seen how you can create applications that combine SCA and servlets. With this combination, you'll use the servlet to expose a web service that can be invoked from a web browser. When the servlet is invoked, it can invoke SCA services by using SCA references. In the next section, we'll look at using a JSP instead of a servlet.

8.2 Writing web component implementations using JSPs

In this section, we'll show how you can use JavaServer Pages (JSPs) to create web component implementations that make use of SCA. This approach is similar to using servlets as component implementations, but you'll use a JSP instead of a servlet. The advantage is that a JSP tends to be easier to write than a servlet.

One of the core differences between JSPs and servlets is that JSPs attempt to separate the application code from the web page. This doesn't mean that there will be no code in the web page. It means that there'll be a clear separation between the two (unlike with servlets where there's only code). Don't worry if this doesn't make much sense now because it'll become clear once you move on to writing your JSP.

When writing a JSP you can make use of tag libraries (sometimes also called taglibs), which provide libraries of code that can be used with your JSP. The JSP specification defines a standard core tag library for tasks such as loops and variables. To simplify using SCA within your JSPs, SCA defines a tag library. The SCA tag library contains a single `reference` tag that can be used to look up SCA services, as shown by the following code fragment:

```
<sca:reference name="myService" type="myservice.MyService"/>
```

This code fragment will look up the SCA reference called `myService` that's defined by the `myservice.MyService` Java interface.

Enough of the theory, let's move on and show how it all works.

8.2.1 Exposing the currency converter using a JSP

In this section, we'll show how the currency converter can be exposed as a web interface using a JSP. It will be based on the example in section 8.1.1, which used a servlet to do the same task. To save duplicating the code, we'll show the differences between using a JSP and a servlet and refer back to the servlet example when the two approaches are the same.

USING TUSCANY IN A JSP

To make use of the SCA taglib, you'll need to include its definition in your JSP. You can then use its tags to look up SCA services. The following listing shows the currency-converter.jsp that makes use of the SCA taglib to invoke the currency converter. It can be found in the contributions/currency-jsp directory of the TuscanySCATours application.

Listing 8.4 Currency converter JSP invoking SCA service

```
<%@ page contentType="text/html;charset=UTF-8"
    language="java" %>
<%@ taglib uri="http://www.osoa.org/sca/sca_jsp.tld"
    prefix="sca" %>
```

❶ Use SCA taglib

```
<sca:reference name="currencyConverter"
    type="com.tuscanyscatours.currencyconverter.
    ➥ CurrencyConverter"/>

<html>
<body>
<h2>SCA Tours Currency Converter JSP</h2>
Welcome to the SCA Tours Currency Converter JSP<p>

<form method=post action="currency-converter.jsp">
Enter value in US Dollars
<input type=text name=dollars size=15><p>
<input type=submit>
</form><p>

<%
   String dollarsStr = request.getParameter( "dollars" );
   if ( dollarsStr != null) {
       double dollars = Double.parseDouble(dollarsStr);
       double converted = currencyConverter.convert(
           "USD", "GBP", dollars);
       out.println(dollars + " US Dollars = "
           + converted + " GB Pounds");
   }
%>
</body>
</html>
```

2 Look up currency converter SCA service

3 Currency input form

4 Convert currency using SCA service

The currency converter JSP defines itself as a JSP and includes the SCA taglib 1. It then uses the reference tag from the SCA taglib to look up the currency converter SCA service 2. The JSP contains a form that's used to enter the amount in U.S. dollars to convert to sterling 3. The final block of JSP code checks whether an amount in U.S. dollars has been entered, and if so, it converts it into sterling using the currency converter service 4.

CONFIGURING THE SCA CURRENCY CONVERTER SERVICE

Your JSP looks up the currency converter SCA service but you haven't yet configured it. Let's do that now by creating a META-INF/sca-deployables/web.composite file that contains the composite definition. You can use the same file as you used for the servlet in listing 8.2 in section 8.1.1.

CONFIGURING THE SERVLET FILTER TO ROUTE REQUESTS TO TUSCANY

The final file that you'll need to define for the JSP is a servlet filter that will direct requests via Tuscany. Once again, you can use the same file as you used for the servlet in listing 8.3 in section 8.1.1, but you'll need to make two changes. You'll need to remove the `<servlet>` and `<servlet-mapping>` tags because you aren't using servlets and change the `<welcome-file-list>` to refer to your JSP. It can be found in the contributions/currency-jsp directory of the TuscanySCATours application and is shown here.

Listing 8.5 Fragment of the servlet filter to direct requests via Tuscany

```
<welcome-file-list>
    <welcome-file>currency-converter.jsp</welcome-file>
</welcome-file-list>
```

Now that you've defined all the files that you need, let's move on to deploying and running the currency converter JSP.

DEPLOYING AND RUNNING THE CURRENCY CONVERTER JSP

Before you can run the currency converter JSP, you'll need to create the WAR file and deploy it on a web server that can host JSPs. Please refer to your web server's documentation for information on how to deploy a WAR file. Once the currency converter WAR file is deployed, you can access the currency converter JSP using a web browser:

```
http://localhost:8080/scatours-contribution-currency-jsp/
```

You should see the currency converter JSP, as shown in figure 8.4.

Figure 8.4 Currency converter JSP viewed through a web browser

Entering 100 in the field and pressing the Submit Query button will cause the currency converter JSP to convert 100 U.S. dollars into sterling using the SCA currency converter service and display the result, as shown in figure 8.5.

As you can see, the JSP has used the currency converter SCA service to do the currency conversion.

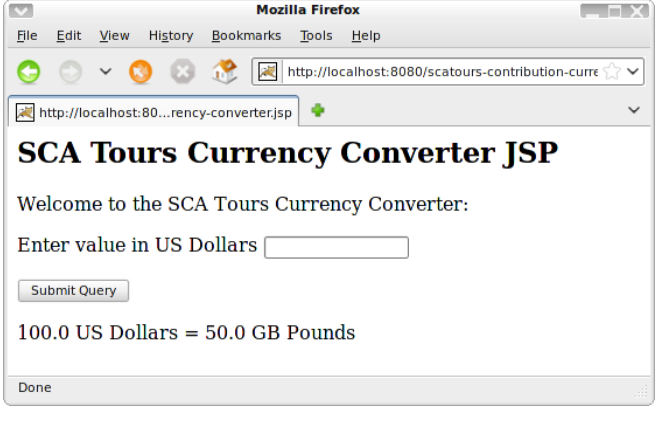


Figure 8.5 Currency converter JSP converted 100 U.S. dollars to sterling.

Let's now move on to looking at how you can use an HTML page as a component implementation.

8.3 HTML pages as SCA component implementations

An alternative approach to using a servlet or JSP to create a web interface is to use an HTML page with JavaScript.

Whereas servlets and JSPs run on the server, HTML and JavaScript run in the browser on the client machine. This has implications for how the Tuscany infrastructure treats these web application artifacts as SCA components.

Regardless of whether you're using servlets, JSPs, or HTML/JavaScript, the artifacts can only be configured with SCA references. This is reasonable because there's no sense that these user interface artifacts provide an SCA service for other computers to use. The service they provide is to the human user.

When an SCA-enabled web container reads the implementation.web element, it's fairly easy to appreciate that the server is able to interact with the servlet or JSP and ensure that the proxy that represents the configured reference is injected into the right place. This is less obvious when using HTML/JavaScript on the client. How does the Tuscany SCA runtime inject reference proxies across the network into the client browser that's running the HTML/JavaScript code?

All will be revealed in this section, where we'll describe how Tuscany supports invoking SCA services that have a JSON-RPC SCA binding using JavaScript contained within HTML pages. The user interface for the TuscanySCATours application uses exactly this approach, and we'll spend some time looking into how it works.

What are JSON and JSON-RPC?

JavaScript Object Notation-Remote Procedure Call (JSON-RPC) is a specification that defines how an application can perform remote procedure calls (RPCs) using HTTP. JSON-RPC will convert the RPC into a HTTP request from the client and send it to the server, where the server can process the request and send back a reply. This is done using standard HTTP. The details of the RPC, such as parameters, are encoded by JSON-RPC into the body of the HTTP request using JavaScript Object Notation (JSON).

JSON is a way of converting data into a text format that's lightweight and independent of a particular computer language. This allows data to be exchanged between applications that may be written in different languages. For example, a JavaScript application could use JSON to send data to a Java application.

8.3.1 Using an HTML page for the TuscanySCATours user interface

Tuscany provides the widget implementation type, implementation.widget, that allows an HTML page to be used as an SCA component implementation.

OVERVIEW OF THE TUSCANYSCATOURS HTML INTERFACE

The TuscanySCATours application makes use of an HTML page for the user interface using implementation.widget. Using this interface, users can search for and book their holidays. Listing 8.6 shows part of the composite XML file for the TuscanySCATours application user interface that uses this implementation.widget element. The full composite XML file can be found in the contributions/fullapp-ui directory of the TuscanySCATours application.

Listing 8.6 TuscanySCATours user interface with implementation.widget

```
<composite xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
    name="fullapp-ui" ...>
  <component name="SCAToursUserInterface">
    <tuscany:implementation.widget location="scatours.html"/>
    <service name="Widget">
      <tuscany:binding.http uri="/scatours"/>
    </service>
    <reference name="scaToursCatalog"
        target="SCATours/SCAToursSearch">
      <tuscany:binding.jsonrpc/>
    </reference>
    ...
  </component>

  <component name="SCAToursComponent">
    <implementation.java class=
        "com.tuscanyscatours.impl.SCAToursImpl"/>
    <service name="SCAToursSearch">
      <tuscany:binding.jsonrpc/>
    </service>
    ...
  </component>
</composite>
```

1 Widget implementation type

2 HTTP binding

3 Catalog SCA reference

4 Catalog SCA service

The TuscanySCATours user interface composite defines an SCAToursUserInterface component that's implemented using the widget implementation type 1. The location attribute is used to specify the name of the HTML file, in this case, scatours.html. Because this isn't a standard SCA implementation type, you'll need to import the Tuscany XML namespace. The SCAToursUserInterface component also defines a service that has an HTTP binding 2 on it. This is so the component can be accessed over HTTP.

The SCAToursUserInterface component also has a reference to the SCAToursSearch service 3 that uses the JSON-RPC binding. This reference will be used by the JavaScript in the HTML to look up the SCA service. There are several other similar references, but we haven't shown them here to keep the example short. The SCAToursSearch service is defined on the SCATours component 4 and has a JSON-RPC binding so it can be invoked from JavaScript in the HTML page.

Let's shift our focus to the scatours.html file to see how it looks up and invokes SCA services. The HTML in the following listing shows the key parts of the scatours.html

file that interacts with the SCAToursSearch service. The full scatours.html file can be found in the contributions/fullapp-ui directory of the TuscanySCATours application.

Listing 8.7 Looking up reference from scatours.html file

```
<html>
<head>
<title>SCA Tours</title>
<link rel="stylesheet" type="text/css" href="style.css" />

<script type="text/javascript" src="scatours.js"></script>

<script language="JavaScript">
  //@Reference
  var scaToursCatalog
    = new tuscany.sca.Reference("scaToursCatalog");

  ...

  function searchTravelCatalog() {
    scaToursCatalog.search(getTripLeg(), search_response);
  }

  function search_response(items, exception) {
    if(exception) {
      alert(exception.javaStack);
      return;
    }
    ...
    for (var i=0; i<items.length; i++) {
      if (items[i].type == "Trip") {

        packagedHTML += '<td>' +  items[i].name + '</td>';
        packagedHTML += '<td>' +  items[i].description + '</td>';
        packagedHTML += '<td>' +  items[i].location + '</td>';
        ...
      }
    }
  }
  ...
}
```

1 Include SCA JavaScript

2 Look up SCA reference

3 Invoke SCA service

4 Handle response

The scatours.html file is read by the browser using the URL http://localhost:8080/scatours/scatours.html. This URL is provided on the server by the HTTP binding associated with the Widget service of the component using implementation.widget. In effect, implementation.widget is presenting the component implementation to the browser and using the SCA composite file to configure the HTML file on the fly with appropriate JSON-RPC reference proxies.

The HTML includes the scatours.js JavaScript file 1 that provides code that can be used to look up SCA references. This JavaScript file is generated automatically by the widget implementation based on the reference configuration of the component using

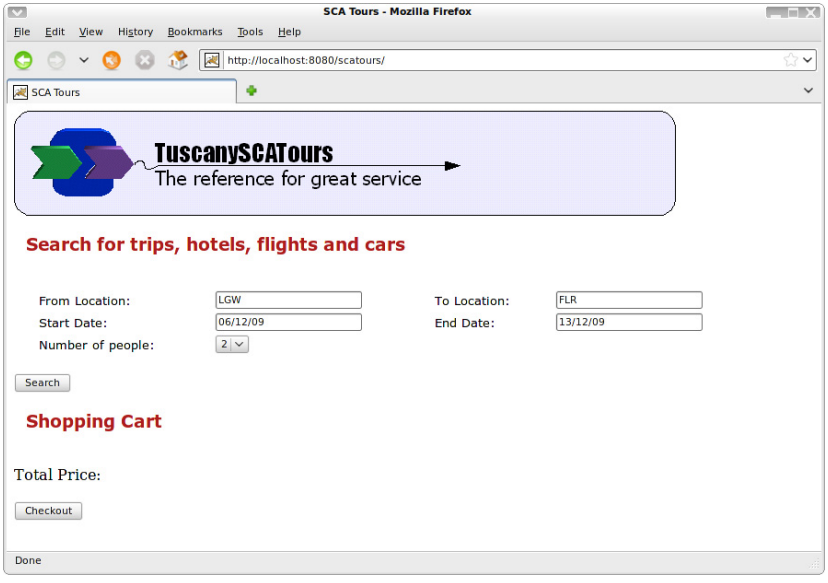


Figure 8.6 The TuscanySCATours HTML page as an SCA component implementation

implementation.widget. The name of the JavaScript file matches the name of the HTML file but with a .js extension rather than .html.

The HTML uses JavaScript to look up an SCA reference ❷ marked with `//@Reference`, which is then used to invoke the `search` operation ❸. The response from the SCA service is returned via the `search_response` JavaScript function ❹, which checks to see if there was an exception and, if not, displays the results on the HTML page.

RUNNING THE TUSCANYSCATOURS HTML INTERFACE

Now that you have an idea of what's required to use an HTML page as an SCA component implementation, let's run the travel application user interface to see it in action using the `launchers/fullapp` launcher from the TuscanySCATours application. Once it starts, use your web browser to access it at http://localhost:8080/scatours/, and you should see a web page similar to the one shown in figure 8.6.

Clicking the Search button will cause the JavaScript in the HTML page to use the SCA reference to do a search for available flights by invoking the `search` method using JSON-RPC and then display a page of the results.

This concludes our time with the implementation.widget. Let's move on to look at exposing file system resources using Tuscany.

8.4 Exposing file system resources

Tuscany provides the ability to publish web resources using SCA components. The component is implemented using implementation.resource, which points at the directory holding the resources. The identified directory is the component implementation.

The HTTP binding, `binding.http`, can then be used to allow HTTP-based browser access to the resources provided by the component implementation. This can be useful when creating a web application using SCA that needs to serve static content such as images.

Here we'll dive straight in and use a set of help pages provided with the travel application to demonstrate how HTML pages on the filesystem resources are exposed to the user's browser via an SCA component.

8.4.1 Exposing the TuscanySCATours help pages

Suppose that, in the TuscanySCATours application, you have some help pages written in HTML. You could deploy a separate web server alongside your SCA application and make the pages available that way. However, that would be no fun because it doesn't use SCA, and this is a book about SCA. The advantage of using SCA to expose the HTML pages is that you can create a single converged application that contains SCA and HTML elements in a single server. If you have a large number of HTML pages, it may be better to use a traditional web server approach, but let's see how it can be done using SCA.

To expose your help pages so that the user can access them using a browser, you'll use the implementation.resource component implementation and `binding.http` in the help-pages.composite file. The help-pages.composite file can be found in the contributions/help-pages directory of the TuscanySCATours application and is shown in the following listing.

Listing 8.8 Exposing the help pages using implementation.resource

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
  targetNamespace="http://www.tuscanyscatours.com"
  name="help-pages">

    <component name="Help">
        <tuscany:implementation.resource
            location="help_pages"/>
        <service name="Resource">
            <tuscany:binding.http
                uri="http://localhost:8085/help/"/>
        </service>
    </component>
</composite>
```

1 Expose help_pages resource

2 Resource service with HTTP binding

The Help component uses implementation.resource as its implementation type 1 and defines that the resources to be exposed are in the help_pages directory. Because implementation.resource is a Tuscany implementation type, the Tuscany XML namespace needs to be defined. Finally, the Resource service 2 is defined with an HTTP binding that specifies that the resources should be published over HTTP at http://localhost:8085/help/.

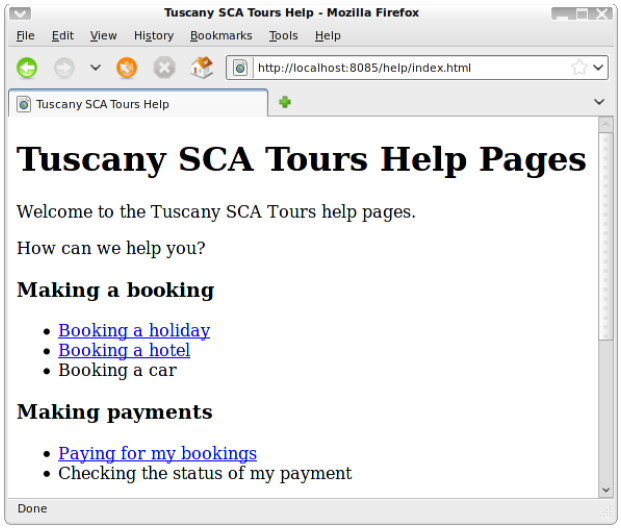


Figure 8.7 Screenshot of the TuscanySCATours help pages

When using implementation.resource, a single Resource service must be defined. This is used by the Tuscany implementation code of implementation.resource to expose the resources over an SCA service. This is why a Resource service is defined in the XML.

Now that you've defined the help pages composite, let's see what happens when you run it. When the launchers/help-pages launcher is run, the help pages will output the following messages to the console:

```
Node started - Press enter to shutdown.
To view the help pages, use your Web browser to view:
    http://localhost:8085/help/index.html
```

If you do as the output suggests and use a web browser to view the help pages at http://localhost:8085/help/index.html, then you should see a web page similar to the one shown in figure 8.7.

Clicking any of the links will result in your browser displaying another help web page. This page will have been served by the Tuscany implementation.resource component implementation. The contents of the HTML pages exist as files in the help_pages directory. If you look there, you'll see several files, including the index.html file displayed in figure 8.7.

These days, users expect to see more than static HTML pages when using web-based applications. They expect to be able to access applications in different ways, including being notified of changes using protocols such as Atom and RSS. Tuscany supports these as well, so let's take a look at these next.

8.5 Exposing component services as Atom and RSS feeds

The use of web feeds has become a popular way for users to be notified of content updates to websites. Users would subscribe to the web feed for the site(s) they're interested in, using their feed-reading software. When the website updates its content, it

updates its web feed listing the updated content. A user's feed-reading software would detect this feed update and inform the user of the updated content. From a user's perspective, using a web feed saves them from having to visit the websites regularly to see if any of the content has been updated.

The two main formats of web feeds are Atom and RSS. They're similar in concept in that they both allow content updates to be published containing information such as title, summary text, and publish date. They both use XML to format the information, but they have different schemas and are therefore not compatible with each other. In this section we'll look at how Tuscany and SCA can be used to publish web feeds in both formats.

8.5.1 Exposing the TuscanySCATours blog as an Atom feed

Suppose that TuscanySCATours wants to set up a corporate blog so that it can inform its customers of the latest news and special offers. Let's see how you can use SCA techniques to expose Atom and RSS feeds using Apache Tuscany.

Apache Tuscany provides support for exposing Atom feeds by adding the Atom binding to a service of a Java component implementation that exposes a feed. We'll examine the implementation code later. First, let's look at the composite file for the TuscanySCATours blog Atom feed. You can find it in the contributions/blog-feed directory of the TuscanySCATours application and in the following listing.

Listing 8.9 TuscanySCATours blog composite exposing an Atom feed

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
  targetNamespace="http://tuscanyscatours.com/"
  name="blogFeed">

  <service name="BlogAtom" promote="BlogFeed">
    <tuscany:binding.atom
      uri="http://localhost:8090/BlogAtom"/>
  </service>

  <component name="BlogFeed">
    <implementation.java class="com.tuscanyscatours.
    ➥ blog.feed.impl.GenericBlogFeedImpl"/>
  </component>
</composite>
```

1 Composite service with Atom binding

2 Blog feed implementation

You'll define a service called BlogAtom for the Blog component 1 and add an Atom binding to it. The Tuscany XML namespace needs to be defined because the Atom binding is a Tuscany binding type. The Atom binding is configured to publish the feed at http://localhost:8090/BlogAtom using the `uri` attribute. Finally, you'll define a Java component implementation called BlogFeed that provides the implementation for the feed 2. You now have a composite file that can be used to expose the TuscanySCATours blog as an Atom feed. One thing we haven't looked at is the BlogFeed component implementation. Let's do that now.

Tuscany supports using the Atom binding on two different types of Java component implementations, namely ones that implement the following:

- org.apache.tuscany.sca.binding.atom.collection.Collection
- The `getAll()` method from the Tuscany `Data` API

The advantages and disadvantages of these two approaches are summarized in table 8.1.

Table 8.1 Comparison of the Atom component implementation types

Implements	Advantage	Disadvantages
`Collection`	You have full control over the Atom feed because you write the code that creates the Atom feed.	You have to write all the Atom feed creation code. Code is tied to Atom feed APIs.
`getAll`	Code is not tied to the Atom feed APIs. Tuscany creates the Atom feed for you.	You have less control over the Atom feed because Tuscany creates it on your behalf.

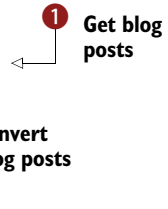
The two approaches are somewhat converse to each other. The simplest approach is to implement the `getAll` method and let Tuscany construct the Atom feed code. This has the benefit of increasing the reusability of the implementation code because other bindings can be applied to it later. However, Tuscany will have made some sensible default choices regarding how the feed will be constructed. These defaults may not match your requirements for how you want the Atom feed to look.

A slightly more complex approach is to implement the `Collection` interface because the implementation code will have to create the Atom feed itself using the Atom APIs. This gives you full control of how the Atom feed will look. But the implementation code is less reusable because it's tied to the Atom APIs.

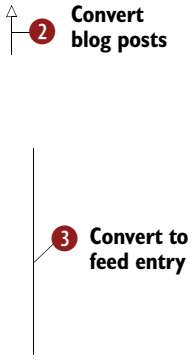
Which approach should you use? It depends on your requirements. Generally, we'd recommend using the `getAll` approach because it's the simplest and most flexible in terms of SCA bindings. But if you know that you'll want to have complete control of how the feed looks, then use the `Collection` approach. For the purposes of this example, we'll use the `getAll` approach, as shown in the following listing. For completeness, there's an example of using the `Collection` approach in class `AtomBlogFeedImpl` in the contributions/blog-feed directory of the TuscanySCATours application.

Listing 8.10 `GenericBlogFeedImpl` exposes the TuscanySCATours blog

```
public class GenericBlogFeedImpl extends BaseBlogFeedImpl {

  public Entry<Object, Object>[] getAll() {
    final List<BlogPost> posts = getAllBlogPosts();

    final Entry<Object, Object>[] entries
      = new Entry[posts.size()];
    int i = 0;
    for (BlogPost post : posts) {
```



```
        entries[i++] = convertBlogPostToFeedItem(post);
    }

    return entries;
  }

  private Entry<Object, Object> convertBlogPostToFeedItem(
      BlogPost post) {
    final Item item = new Item(post.getTitle(),
      post.getContent(), post.getLink(),
      post.getRelated(), post.getUpdated());
    final Entry<Object, Object> entry
      = new Entry<Object, Object>(nextBlogID(), item);
    return entry;
  }
```



The `GenericBlogFeedImpl` class implements the `getAll` method from the Tuscany `Data` API. This means that the SCA Atom binding will manage the creation of the Atom feed and the conversion of the Tuscany `Data` items to Atom feed entries. The `getAll` method gets all blog posts using the `getAllBlogPosts` method (1), which returns the list of blog posts in some internal format. The implementation for the `getAllBlogPosts` method isn't shown here because it's implemented by the `BaseBlogFeedImpl` superclass. In this example, it returns a hardcoded list of blog posts. In a real-world application, it would load the blog posts from some persistent storage such as a database or filesystem.

Now that you have all the blog posts, you'll need to convert them to instances of the `Entry` class from the Tuscany `Data` API (2) by calling the `convertBlogPostToFeedItem` method (3) on each blog post. The Tuscany `Entry` class provides a feed technology independent implementation of a feed item.

Now that you have your composite file and implementation code for the TuscanySCATours blog feed, run it using the `launchers/blog-feed` launcher from the TuscanySCATours application and see if you can access the Atom feed. When run, it will output the following messages to the console:

```
Node started - Press enter to shutdown.
To view the blog feed, use your Web browser to view:
    http://localhost:8090/BlogAtom
```

Follow the instructions and see if you can access the Atom feed by accessing the address http://localhost:8090/BlogAtom using a web browser. You'll see the web page shown in figure 8.8.

With Mozilla Firefox, you're presented with an option to subscribe to the feed. If you click the blog entry link, you'll be redirected to the *Tuscany SCA in Action* website.

How can you verify that this is indeed an Atom feed? The easiest way is to examine the source for the page (in Mozilla Firefox, right-click View Page Source). If you do, you'll see the Atom XML header, a fragment of which is shown here:

```
<?xml version='1.0' encoding='UTF-8'?>
<feed xmlns="http://www.w3.org/2005/Atom">
```

That confirms it; you've exposed the TuscanySCATours blog as an Atom feed.

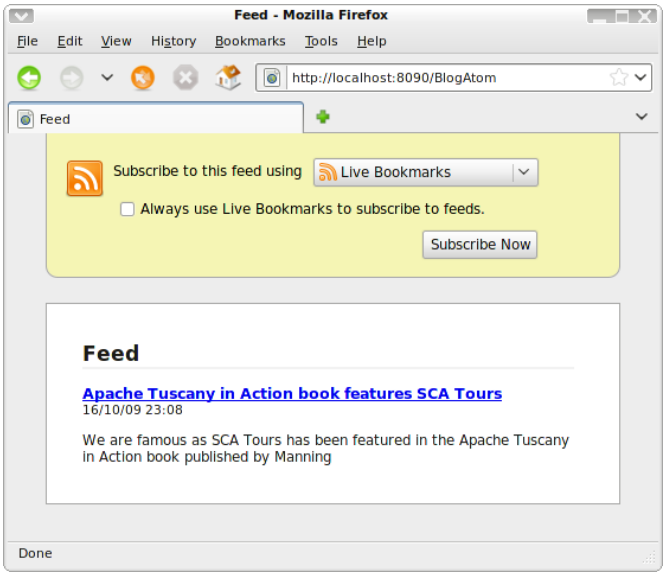


Figure 8.8 Screenshot of the TuscanySCATours blog Atom feed

8.5.2 Extending the TuscanySCATours blog with an RSS feed

Now that you've implemented the Atom feed for the TuscanySCATours blog, you'll add an RSS feed to it. Because you implemented your blog feed code by implementing the `getAll` method and are using Tuscany to convert the entries to feed entries, you'll need to add a new service with a Tuscany RSS binding to the blog-feed.composite XML file. It can be found in the contributions/blog-feed directory of the TuscanySCATours application and is shown here.

Listing 8.11 TuscanySCATours blog composite exposing an Atom feed and an RSS feed

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
  targetNamespace="http://goodvaluetrips.com/"
  name="blogFeed">

  <service name="BlogAtom" promote="BlogFeed">
    <tuscany:binding.atom
      uri="http://localhost:8090/BlogAtom"/>
  </service>
  <service name="BlogRSS" promote="BlogFeed">
    <tuscany:binding.rss
      uri="http://localhost:8090/BlogRSS"/>
  </service>

  <component name="BlogFeed">
    <implementation.java
      class="com.tuscanyscatours.blog.feed.impl.GenericBlogFeedImpl"/>
  </component>
</composite>
```

1 Service with RSS binding

2 RSS binding

All we needed to do was add another service for the RSS feed ❶ and add the RSS binding to it ❷. Simple as that—no code changes are required! But did you notice that we added another composite service rather than adding the RSS binding to the existing BlogAtom service? The reason for this is to show that it's simple to promote the same service with a different name and binding. We could have just added the RSS binding to the existing BlogAtom service and accepted that the BlogAtom service has both Atom and RSS bindings.

If we had decided to take the other approach and coded the TuscanySCATours blog so that it extended the `org.apache.tuscany.sca.binding.atom.collection.Collection` interface, we'd need to write a new Java component implementation that extended the `org.apache.tuscany.sca.binding.rss.collection.Collection` interface because the two interfaces aren't compatible. For completeness, there's an example of using the `Collections` approach in class `RSSBlogFeedImpl` in the TuscanySCATours application.

Running the TuscanySCATours blog feed again using the launchers/blog-feed launcher of the TuscanySCATours application will output the following messages to the console:

```
Node started - Press enter to shutdown.
To view the blog feed, use your Web browser to view:
    http://localhost:8090/BlogAtom
    http://localhost:8090/BlogRSS
```

Follow the instructions and see if you can access the RSS feed by accessing the address http://localhost:8090/BlogRSS using a web browser. You'll see the web page shown in figure 8.9.

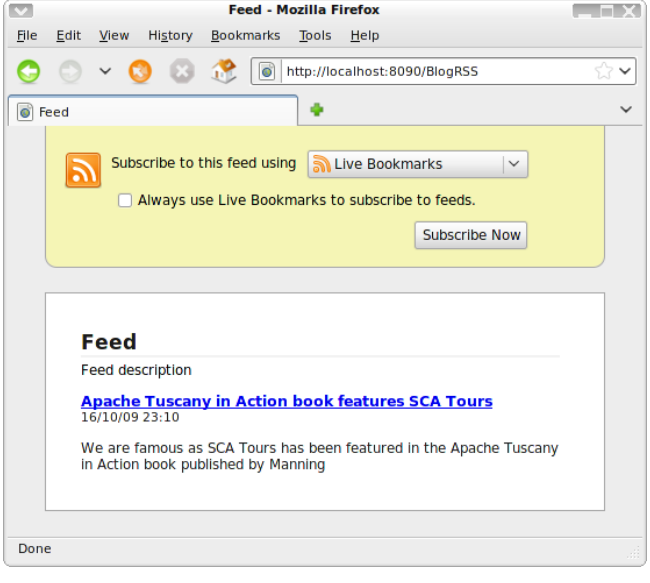


Figure 8.9 Screenshot of the TuscanySCATours blog RSS feed

With Mozilla Firefox, you're presented with an option to subscribe to the feed. If you click the blog entry link, you'll be redirected to the *Tuscany in Action* website.

How can you verify that this is indeed an RSS feed? The easiest way is to examine the source for the page (in Mozilla Firefox, right-click View Page Source). If you do, you'll see the RSS XML header, a fragment of which follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:content="http://purl.org/rss/1.0/modules/content/"
```

This shows that you've exposed the TuscanySCATours blog as an RSS feed.

You've seen that it's straightforward to construct a component so that it can provide an Atom feed. It's similarly straightforward to expose a component service as an RSS feed. The Atom and RSS bindings can be used independently but will often but used together to provide access to a component service to the maximum number of users. You'll need to think about this because it has an impact on how you'll construct your component implementation.

Now that you know how to expose component services using Atom or RSS, let's look at how an SCA component can read an Atom or RSS feed.

8.6 Referencing Atom and RSS feeds

In the previous section, we looked at how you could expose component services as Atom and RSS feeds. We'll now move on to look at how you can use Atom and RSS feeds in your SCA applications. Fortunately, Tuscany provides support for accessing Atom and RSS feeds using `binding.atom` and `binding.rss`, respectively.

8.6.1 Logging the TuscanySCATours blog Atom feed

As you saw earlier, TuscanySCATours now has Atom and RSS feeds for their blog. Being the professional organization that they are, they decide they should keep a log of the Atom and RSS feeds so they can record what feed entries are posted. For this purpose, they decide to write the TuscanySCATours feed logger to log the feeds.

To simplify the running of this example, rather than running your own feeds, you're going to use the Atom and RSS feeds from the mocked-up TuscanySCATours blog that can be found at http://scatours.wordpress.com.

The first thing you'll need to access the TuscanySCATours Atom feed is the URL of where it's published: http://scatours.wordpress.com/?feed=atom. To make sure that it's working correctly, it's probably worth attempting to access the feed URL in your web browser. Now that you know the URL of the feed, let's have a look at the composite file for the TuscanySCATours feed logger. It can be found in the feed-logger.composite file in the contributions/feed-logger directory of the TuscanySCATours application and is shown in the following listing.

Listing 8.12 TuscanySCATours feed logger composite accessing an Atom feed

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
  targetNamespace="http://tuscanyscatours.com/"
  name="feedLogger">
```

```
  <service name="FeedLogger" promote="FeedLogger">
  </service>

  <component name="FeedLogger">
    <implementation.java class="com.tuscanyscatours.
    ➥ feedlogger.impl.FeedLoggerImpl"/>
    <reference name="scaToursBlogAtom">
      <tuscany:binding.atom
        uri="http://scatours.wordpress.com/?feed=atom"/>
    </reference>
  </component>
</composite>
```

1 **Define FeedLogger service**

2 **FeedLogger component**

3 **SCA Tours Atom reference**

The XML then defines the FeedLogger service 1 that exposes the FeedLogger component. You'll use this later to invoke the TuscanySCATours feed logger. The FeedLogger component is a Java component implementation 2 and has a single reference with a `<binding.atom>` element 3 that uses the Atom binding to the TuscanySCATours Atom feed. Because `binding.atom` is a Tuscany binding, you must define the Tuscany XML namespace.

It's time to turn your attention to the TuscanySCATours feed logger implementation code. The previous composite used the `FeedLoggerImpl` class as the Java implementation. The code for this Java class is shown in the following listing. It can also be found in the contributions/feed-logger directory of the TuscanySCATours application.

Listing 8.13 Fragment of the `FeedLoggerImpl` class

```
public class FeedLoggerImpl implements FeedLogger {

  @Reference
  public org.apache.tuscany.sca.binding.atom.collection.Collection
     scaToursBlogAtom;

  public void logFeeds(int maxEntriesPerFeed) {
    System.out.println("Logging SCA Tours Blog Atom feed:");
    logAtomFeed(scaToursBlogAtom, maxEntriesPerFeed);
  }

  private void logAtomFeed(org.apache.tuscany.sca.binding.
  ➥ atom.collection.Collection atomFeed, int maxEntries) {
    final Feed feed = atomFeed.getFeed();
    System.out.println("Feed: " + feed.getTitle());
    final List<Entry> entries = feed.getEntries();

    for (int i = 0; i < entries.size() && i < maxEntries; i++) {
      Entry entry = entries.get(i);
      System.out.println("Entry: " + entry.getTitle());
    }
    System.out.println();
  }
}
```

1 **Injected SCA Tours Atom feed reference**

2 **Get Atom feed entries**

3 **Log Atom feed entries**

The `FeedLoggerImpl` class has an injected reference to the TuscanySCATours Atom feed 1 that it uses to retrieve all the feed entries 2 and log their titles 3 to the console.

Now that your composite and Java component implementation are complete, try running the TuscanySCATours feed logger using the `launchers/feed-logger` launcher from the TuscanySCATours application. When run, the TuscanySCATours feed logger will output the following messages to the console:

```
Logging SCA Tours Blog Atom feed:
Feed: SCA Tours Blog
Entry: Apache Tuscany in Action
Entry: Hello and welcome to SCA Tours
```

From this output, you can see that the TuscanySCATours feed logger has pulled two entries off the Atom feed. Note that when you run the TuscanySCATours feed logger, the titles of the feed entries may be different if newer entries have been posted to the TuscanySCATours blog.

8.6.2 Logging the TuscanySCATours blog RSS feed

It's now time to turn your attention to adding the TuscanySCATours RSS feed to the TuscanySCATours feed logger. The URL for this RSS feed is http://scatours.wordpress.com/?feed=rss. To make sure that it's working correctly, it's probably worth attempting to access the feed URL in your web browser. Update the TuscanySCATours feed logger composite XML file to include the TuscanySCATours blog RSS feed. It can be found in the feed-logger.composite file in the contributions/feed-logger directory of the TuscanySCATours application and is shown here.

Listing 8.14 TuscanySCATours feed logger composite accessing Atom and RSS feeds

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
  targetNamespace="http://tuscanyscatours.com/"
  name="feedLogger">

  <service name="FeedLogger" promote="FeedLogger">
  </service>

  <component name="FeedLogger"
    <implementation.java class="com.tuscanyscatours.feedlogger.impl.
    ➥ FeedLoggerImpl"/>
    <reference name="scaToursBlogAtom">
      <tuscany:binding.atom
        uri="http://scatours.wordpress.com/?feed=atom"/>
    </reference>
    <reference name="scaToursBlogRSS">
      <tuscany:binding.rss
        uri="http://scatours.wordpress.com/?feed=rss"/>
    </reference>
  </component>
</composite>
```

1 SCA Tours RSS reference

2 RSS binding

In this composite, you've added a new reference called `scaToursBlogRSS` 1 and added an RSS binding to it that points at the TuscanySCATours blog RSS feed 2.

Let's update the TuscanySCATours feed logger implementation code to use the new reference to log the RSS feed. Unfortunately, Tuscany currently doesn't support

generic consumption of feeds via the Tuscany Data API. But support for it may be added in later versions of Tuscany. Therefore, you'll need to update the code in the `FeedLoggerImpl` class to handle the RSS feed, as shown here. The full source code can be found in the contributions/feed-logger directory of TuscanySCATours.

Listing 8.15 Updated the `FeedLoggerImpl` class with RSS logging added

```
public class FeedLoggerImpl implements FeedLogger {

  @Reference
  public org.apache.tuscany.sca.binding.atom.collection.Collection
     scaToursBlogAtom;

  @Reference
  public org.apache.tuscany.sca.binding.rss.collection.Collection
     scaToursBlogRSS;

  public void logFeeds(int maxEntriesPerFeed) {
    System.out.println("Logging SCA Tours Blog Atom feed:");
    logAtomFeed(scaToursBlogAtom, maxEntriesPerFeed);

    System.out.println("Logging SCA Tours Blog RSS feed:");
    logRSSFeed(scaToursBlogRSS, maxEntriesPerFeed);
  }

  private void logAtomFeed(org.apache.tuscany.sca.binding.
  ➥ atom.collection.Collection atomFeed, int maxEntries) {
  ...
  }

  private void logRSSFeed(org.apache.tuscany.sca.binding.
  ➥ rss.collection.Collection rssFeed, int maxEntries) {
    SyndFeed feed = rssFeed.getFeed();
    System.out.println("Feed: " + feed.getTitle());
    List<SyndEntry> entries = feed.getEntries();

    for (int i = 0; i < entries.size() && i < maxEntries; i++) {
      SyndEntry entry = entries.get(i);
      System.out.println("Entry: " + entry.getTitle());
    }
    System.out.println();
  }
}
```

1 **Injected SCA Tours RSS feed reference**

2 **Get RSS feed entries**

3 **Log RSS feed entries**

The updated `FeedLoggerImpl` class now has an additional injected reference to the TuscanySCATours RSS feed 1 that it uses to retrieve all the feed entries 2 and log their titles 3.

Run the TuscanySCATours feed logger again using the `launchers/feed-logger` launcher from the TuscanySCATours application. This time, you'll get the following messages to the console:

```
Logging SCA Tours Blog Atom feed:
Feed: SCA Tours Blog
Entry: Apache Tuscany in Action
Entry: Hello and welcome to SCA Tours
```

```
Logging SCA Tours Blog RSS feed:
Feed: SCA Tours Blog
Entry: Apache Tuscany in Action
Entry: Hello and welcome to SCA Tours
```

From this output, you can see that the TuscanySCATours feed logger has pulled the same entries from the TuscanySCATours blog over the Atom feed and the RSS feed. Simple, wasn't it?

8.7 Summary

In this chapter, you saw how you can create web clients in SCA using a variety of technologies including servlets, JSPs, JSON-RPC, Atom, and RSS feeds. The various SCA implementation types and bindings supported by Tuscany are able to simplify creating these web clients by hiding some of the complexities.

You've been able to develop both client-side (HTML/JavaScript) and server-side (servlet/JSP) user interface components that make use of the same backend SCA components. The SCA composite model presents a coherent picture of the client and server while hiding some of the complexities of how they're wired together.

Over the last two chapters, you've seen how different technologies can be wired together with SCA. It's time to move to the next chapter and see how SCA can help in transforming data between technologies that use different vocabularies.

9 Data representation and transformation

This chapter covers

- Describing and representing data in various ways
- Explaining requirements for data representations
- Transforming data without application coding

Handling of data in a service-oriented environment can be challenging. Data has to travel over the network between service providers and consumers, and this data exchange may be implemented using different technologies. In this chapter we'll explain how data can be represented, where data transformation is needed, and how Tuscany handles this in order to preserve the flexibility of SCA-enabled applications.

We'll use the credit card payment example from the TuscanySCATours scenario to demonstrate how collaborating business services deal with data exchange. The example can be found in the code at the following locations:

- contributions/payment-java
- contributions/creditcard-payment-sdo
- contributions/databinding-client
- launchers/databinding

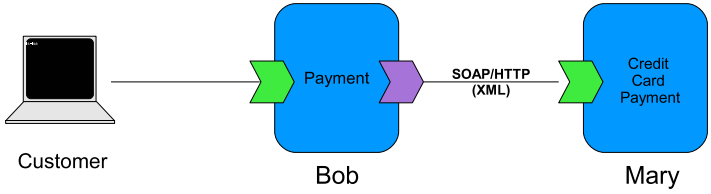


Figure 9.1 The Payment and CreditCardPayment components that Bob and Mary are working on

As illustrated in figure 9.1, there are two related business components: Payment and CreditCardPayment. In this case, the Payment component takes credit card information submitted directly by the customer over the internet. When payment is made via a credit card, the Payment component talks to the CreditCardPayment component to authorize the charge. Two developers, Bob and Mary, are responsible for getting these two components to work together. Bob is going to implement the Payment service while Mary develops the CreditCardPayment service.

In the next sections, we'll follow Bob and Mary and learn how they assemble the Payment and CreditCardPayment components.

We'll start by looking at how a service interface is used to describe what data will flow between a component reference and a component service. Then we'll look at how the data described in the service interface can be represented and accessed within the running component implementation. We'll use JAXB and SDO as examples of Java data representations.

JAXB

JAXB (Java Architecture for XML Binding) is defined by JSR 222 (http://jcp.org/en/jsr/detail?id=222). It provides a convenient way to process XML content using Java objects by binding its XML schema to Java representation. At the time of writing, Tuscany uses the JAXB version 2.1.7 implementation from Sun.

SDO

SDO (Service Data Objects) is designed to simplify and unify the way in which applications handle data. Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, web services, and enterprise information systems (http://www.osoa.org/display/Main/Service+Data+Objects+Specifications). At the time of writing, Tuscany uses its own version 1.1.1 implementation of SDO.

SCA allows interface contracts between references and services to be specified and controlled in a number of ways, so we'll look at that next before finally moving on to look at why data transformations are important and how the Tuscany SCA databinding

framework can perform many data transformations automatically without requiring explicit code in the component implementation.

9.1 Data exchange between SCA components

In this first section we'll follow Bob and Mary as they describe the interface for the CreditCardPayment component service. As you build and describe your components in SCA, one of the most important things that you'll do in terms of controlling the way that data is represented is to describe the component service interfaces. Service interfaces are often referred to as *contracts*, and we'll use both terms here. You'll see in this section that Bob and Mary choose WSDL 1.1 as a common, standard, and interoperable interface description language. The WSDL interface they describe is referenced directly from the composite file that describes their application.

The first thing Bob and Mary do is agree on what information should be exchanged between the Payment and CreditCardPayment components. In our example, the Payment component will provide the CreditCardPayment component with detailed information about the credit card and the amount of charge. The CreditCardPayment component will then respond with the status of the card transaction. With some help from the business analysts, Bob and Mary find out that the authorize operation should exchange the following information about a credit card transaction, shown in table 9.1.

Table 9.1 An informal description of the data exchange required for the CreditCardPayment component authorize operation

Operation: authorize	Data description
Input	Credit card detail: Credit card type: MasterCard, Visa, Amex, Discover Credit card number Expiration month and year Credit card verification code Card owner Name Address Street City State Zip Code Home phone Amount of charge
Output	Status of the card transaction

The description of the business operation becomes a contract to bind the Payment component as a service consumer to the CreditCardPayment component as the service provider. Table 9.1 defines the necessary parts of the input/output data that Bob and Mary agreed to. This information will ensure that the CreditCardPayment

component has the data required to authorize credit card charges requested by the Payment component.

To formally describe the business functions and data exchange requirements, Bob and Mary now need to define a business interface for the CreditCardPayment component. SCA supports several ways to define service interfaces, such as Java interfaces, WSDL 1.1 port type, WSDL 2.0 interface, or CORBA IDL. They all have constructs to define the operations as well as the data structures and types.

9.1.1 Using WSDL to describe the CreditCardPayment interface

Interoperability across different systems is important to Mary and Bob, so they choose to use WSDL 1.1 and XML Schema 1.0 to define the CreditCardPayment interface and the CreditCardDetails data type. WSDL 1.1 is a W3C standard and is in common use, so it meets their requirement for interoperability.

> **Interface/data decoupling**
>
> Using a programming language–neutral interface definition such as WSDL/XSD promotes loose coupling between the service consumer and provider. This happens because the data exchange is document oriented and programming language independent. By contrast, component interfaces that expose programming language–specific forms of data (for example, Java) promote tight coupling, because this usually forces the use of the same programming language on each side of the interface.

Listing 9.1 shows the WSDL definition that describes the CreditCardPayment interface and its authorize operation. This WSDL could have been written by hand, or it could have been generated using your favorite WSDL tool such as the WSDL editor from the Eclipse Web Tools Platform (WTP) project.

Listing 9.1 The WSDL definition for the CreditCardPayment service interface

```
<wsdl:definitions name="CreditCardPayment"
  targetNamespace="http://www.example.org/CreditCardPayment/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://www.example.org/CreditCardPayment/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

  <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              targetNamespace="http://www.example.org/CreditCardPayment/"
                xmlns:tns="http://www.example.org/CreditCardPayment/">

      <xsd:element name="authorize" type="tns:AuthorizeType" />

      <xsd:complexType name="AuthorizeType">
        <xsd:sequence>
```

Data structure description

```
    <xsd:element name="CreditCard"
                 type="tns:CreditCardDetailsType"/>
    <xsd:element name="Amount" type="xsd:float"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="authorizeResponse"
             type="tns:AuthorizeResponseType" />

<xsd:complexType name="AuthorizeResponseType">
  <xsd:sequence>
    <xsd:element name="Status" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="CreditCardDetailsType">
  <xsd:sequence>
    <xsd:element name="CreditCardType"
                 type="tns:CreditCardTypeType" minOccurs="0" />
    <xsd:element name="CreditCardNumber"
                 type="xsd:string" minOccurs="0" />
    <xsd:element name="ExpMonth" type="xsd:int"
                 minOccurs="0" />
    <xsd:element name="ExpYear" type="xsd:int"
                 minOccurs="0" />
    <xsd:element name="CardOwner" type="tns:PayerType"
                 minOccurs="0" />
    <xsd:element name="CVV2" type="xsd:string"
                 minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="CreditCardTypeType">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="Visa" />
    <xsd:enumeration value="MasterCard" />
    <xsd:enumeration value="Discover" />
    <xsd:enumeration value="Amex" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="PayerType">
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string" />
    <xsd:element name="Address" type="tns:AddressType" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="AddressType">
  <xsd:sequence>
    <xsd:element name="Street" type="xsd:string" />
    <xsd:element name="City" type="xsd:string" />
    <xsd:element name="State" type="xsd:string" />
    <xsd:element name="ZipCode" type="xsd:string" />
    <xsd:element name="HomePhone" type="xsd:string" />
```

Data structure description

```
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="AuthorizeRequest">
    <wsdl:part name="parameters" element="tns:authorize"/>
  </wsdl:message>

  <wsdl:message name="AuthorizeResponse">
    <wsdl:part name="parameters" element="tns:authorizeResponse"/>
  </wsdl:message>

  <wsdl:portType name="CreditCardPayment">
    <wsdl:operation name="authorize">
      <wsdl:input message="tns:AuthorizeRequest"/>
      <wsdl:output message="tns:AuthorizeResponse"/>
    </wsdl:operation>
  </wsdl:portType>
...

</wsdl:definitions>
```

Collect data types into messages

Reference input/output messages

Binding and service elements omitted for readability

If you look toward the top of the WSDL document in listing 9.1, you'll see a WSDL `<wsdl:types>` element, which contains a set of the XML schema types. These schema types describe the structure of the data flowing in and out of the authorize operation. The authorize operation itself is defined by a WSDL `<wsdl:operation name="authorize">` element within the WSDL `<wsdl:portType name="CreditCardPayment">`. You can also see that WSDL `<wsdl:message>` elements referenced by the authorize operation map the schema types to the input and output messages of the operation.

Figure 9.2 shows a graphical visualization of the XML schema types contained in the `<wsdl:types>` section of the WSDL document. It matches the informal description of the data Bob and Mary collected in table 9.1.

As you can see from the XML schema in listing 9.1 and from its visual representation in figure 9.2, the data required to describe the customer's credit card comprises several nested levels of data fields. In order to access and manipulate this data within a component implementation, you'll need a suitable data structure.

It's often the case when describing component service interfaces that you create the WSDL definition containing a portType, use the WSDL as part of the component description in your composite file, and then use the WSDL to generate data structures

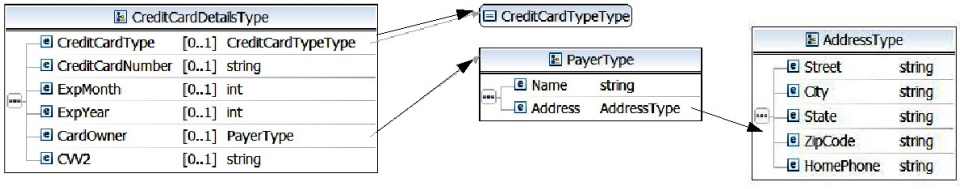


Figure 9.2 Visualization of the XML schema definition for CreditCardDetailsType

for use in your business logic. This is often called a top-down approach. You're starting from abstract WSDL and using it to generate the concrete interfaces and structures used in your component implementation.

Alternatively, it may be the case that you're starting from a component implementation, for example, a Java class that implements a Java interface. In this case the Java interface is supported by SCA, and you may not need to worry about WSDL at all. This often happens if you're implementing all of your components using Java. If WSDL is required, because you want to present your service interfaces to remote clients that require it, then the Tuscany runtime can generate the WSDL for you based on your component description. If you have a component service that uses `binding.ws`, you can generate a WSDL description of the component service by pointing your browser at the service endpoint URL with ?wsdl added to the end, for example, http://somehost:8080/MyComponentService?wsdl.

In this chapter we'll be taking the top-down approach and defining the WSDL to start with because it helps us show how different data formats can be used. You'll see in section 9.2 how the data structures can be generated from the WSDL and used in the component implementation. Before we get to building the component, let's see how Bob and Mary use the WSDL interface description in an SCA composite file.

9.1.2 Using WSDL in an SCA composite

Bob and Mary have now reached an agreement on how to exchange data between Payment and CreditCardPayment components, and a WSDL file has been constructed describing the contract. If they were to build an SCA composite file now to describe the wiring between the two components, it would look something like the following composite file.

Listing 9.2 Using WSDL interfaces in component references and services

```
<composite ...>
  <component name="Payment">
<implementation.java
class="com.tuscanyscatours.payment.impl.PaymentImpl"/>
 <reference name="creditCardPayment">
   <binding.ws wsdlElement = "http://www.example.org/CreditCardPayment/
   ➥ wsdl.port(CreditCardPaymentService/CreditCardPaymentPort)"
 </reference>
</component>

<component name="CreditCardPayment">
  <implementation.java
class="com.tuscanyscatours.payment.creditcard.impl.CreditCardPaymentImpl"/>
  <service name="CreditCardPayment">
    <interface.wsdl interface = "http://www.example.org/CreditCardPayment/
    ➥ wsdl.interface(CreditCardPayment)"
    <binding.ws uri="http://localhost:8081/CreditCardPayment" />
  </service>
</component>
</composite>
```

Binding configuration using WSDL (1)

Interface defined using WSDL (2)

In the Payment component `creditCardPayment` reference definition, the `<binding.ws>` element (see chapter 7 for the syntax) references the WSDL service port ❶. The Tuscany `binding.ws` implementation is able to read the WSDL document and extract enough information about the target interface and its network location to make the call to the CreditCardPayment component.

In the CreditCardPayment component `CreditCardPayment` service definition, a slightly different approach is taken. The WSDL document is referenced in the `<interface.wsdl>` element by QName of the portType ❷, and the `<binding.ws>` element is configured manually with the URI of the endpoint that's to be exposed by the service. The WSDL service and binding will be generated using SOAP document literal style.

Why does SCA support these two different approaches? The WSDL document is a generic mechanism for describing service interfaces and isn't always bound directly to the Web Services binding. So when defining the CreditCardPayment service on the CreditCardPayment component, we could have used another type of binding, for example, `binding.jms`, but still use `interface.wsdl` to allow the WSDL document to describe the structure of the service interface. For those of you familiar with Web Services, it may seem a little odd that WSDL can be used with anything other than SOAP/HTTP Web Services. WSDL does a good job in describing the shape of an interface independently of its binding, so it does work well in this situation.

You may be wondering how the Tuscany SCA runtime is able to find the WSDL file when this composite file references the WSDL elements such as port type or service only by XML qualified name and not by physical location. Remember that the Tuscany runtime reads contributions, which contain composite files, Java classes, and WSDL files, among other things. When Tuscany first reads the contribution, it indexes all the artifacts it finds. Hence, the application developer doesn't need to provide the physical location of the WSDL file. As long as the WSDL file is included in the contribution, Tuscany can find it automatically.

Now that Bob and Mary have described the contract between their two components, using WSDL, they'll implement the business logic using SCA Java components. Bob must create the `PaymentImpl` class and Mary must create the `CreditCardPaymentImpl` class, and they must each choose a data representation technology. Even though WSDL is used to formally define the data exchange between the two components over the Web Services protocol, it isn't natural to work from WSDL and XSD when writing Java programs. It's much more convenient to have Java classes that represent the data structures.

Bob and Mary will map the WSDL form of the service interface into a Java interface. The resulting interface will represent operation parameters and return values using Java types. These types are the in-memory representation of the data being exchanged between components. Let's now look at the Java-based technologies Bob and Mary chose to represent the data.

9.2 Representing data within component implementations

Data needs to have an in-memory representation so that component implementation logic can access and manipulate it. It's natural that the component developer will choose an appropriate representation of the business interface's business data to facilitate data processing. We'll first look at how Bob exploits JAXB to interact with a reference to the CreditCardPayment component. Then we'll look at how Mary uses SDO to implement the CreditCardPayment authorize operation.

Although Bob and Mary have chosen Java technology to represent the credit card data in their component implementations, the SCA Web Services binding, `binding.ws`, needs to handle the data and marshal and unmarshal it to and from SOAP envelopes that pass over HTTP. The Tuscany implementation of the Web Services binding is based on technology from the Apache Axis2 project (http://ws.apache.org/axis2/). The Web Services binding interface receives and produces data in the form of Axis AXIOM objects.

AXIOM

AXIOM stands for Axis Object Model (also known as OM, or Object Model) and refers to the XML Infoset model that was initially developed for Apache Axis2. For more information, please see http://ws.apache.org/commons/axiom/.

Figure 9.3 gives an overview of the different data representations involved in the various parts of our simple scenario for the purpose of illustrating data exchange in a typical composite application. The data is represented as JAXB objects at the Payment component reference. It's then converted into an Axiom XML model so that the Axis2

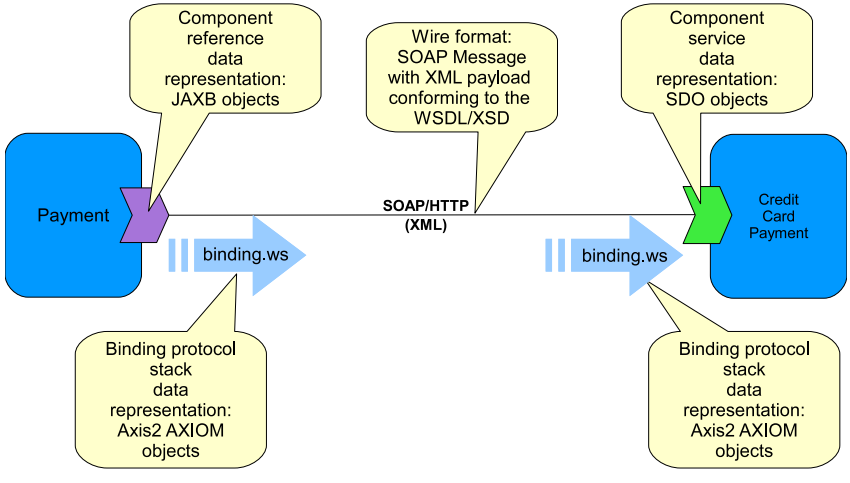


Figure 9.3 As a message passes from component reference to component service, its data is represented in several different formats depending on whether the message is in the component, in the binding, or being passed on the wire.

stack behind the Web Services binding can send it out in a SOAP message. On the wire, the data is an XML document inside a SOAP envelope. At the service ends, the reverse happens; the data is read into an AXIOM model and then is converted into SDO objects before being passed to the CreditCardPayment service.

You'll see later on how the Tuscany runtime arranges for the data to be transformed between these various formats. First though, let's look at how Bob and Mary deal with the data in their component implementations.

9.2.1 Passing data to component references using JAXB objects

Remember that Bob is implementing the Payment component that references the CreditCardPayment component. Bob is a Java developer, and he wants to implement the payment service in Java. To talk to the CreditCardPayment service, which we know is described using a WSDL port type, Bob prefers to have a Java interface, which is compatible with the WSDL port type. Furthermore, there are many different ways to represent the CreditCardDetails data type in Java, for example:

- Raw data, such as `byte[]`, `String`, `InputStream`, or `Reader`
- XML parsing technologies, such as DOM, SAX events, StAX
- Java/XML binding technologies, such as JavaBeans, JAXB, SDO (generated static SDO classes or dynamic DataObject), AXIOM

Bob has a few requirements in mind for the data representation:

- Strong-typed access for the business information through methods such as `getAddress()`
- Can be used to produce XML documents
- Can be used to hold request data from the web frontend

To talk to a web service using Java, Bob needs to create an interface with methods that map to the set of WSDL operations. He also needs to have Java classes to represent the data types defined by the XML schema. Bob could write the plain Java classes by hand, but some of the WSDL and XSD information such as the namespace and element names would be lost. As a result, the Java classes can't be mapped to WSDL precisely.

To simplify Java programming with Web Services without dealing with the complexity of WSDL and XSD, the SCA specification recommends JAX-WS to map the WSDL interface into a Java interface. Bob adopts JAX-WS and also chooses JAXB as the representation of XML data. Both JAX-WS and JAXB use Java annotations to customize Java classes with the complete metadata from the WSDL and XSD.

JAX-WS

JAX-WS stands for Java API for XML Web Services. It defines rules and Java annotations to map Web Services to Java and vice versa. You can read the specification at http://jcp.org/en/jsr/detail?id=224.

JAX-WS and JAXB are both part of JDK 6 update 6 or later. Standalone distributions can also be downloaded from the web. Bob runs the JAX-WS wsimport tool from his JDK to generate the Java interfaces and corresponding JAXB classes from the WSDL and XSD. Here's how that's done:

```
wsimport -p com.tuscanyscatours.payment.creditcard CreditCardPayment.wsdl
```

Instead of using the tools provided by JDK 6 or other JAX-WS implementation projects, Bob could have used the JAX-WS wsimport Maven plug-in or ANT tasks to generate the Java classes. We use Maven in the TuscanySCATours example, and the following is included in the payment-java contribution pom.xml file.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxws-maven-plugin</artifactId>
  <version>1.9</version>
  <executions>
    <execution>
      <id>wsimport</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>wsimport</goal>
      </goals>
      <configuration>
        <packageName>com.tuscanyscatours.payment.creditcard</packageName>
        <wsdlDirectory>${basedir}/src/main/resources</wsdlDirectory>
        <wsdlFiles>
          <wsdlFile>CreditCardPayment.wsdl</wsdlFile>
        </wsdlFiles>
        <sourceDestDir>
            ${project.build.directory}/jaxws-source
        </sourceDestDir>
        <verbose>false</verbose>
        <xnocompile>true</xnocompile>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The wsimport command or plug-in will generate the JAX-WS service endpoint interfaces as well as the JAXB classes for the complex XSD types. The list of generated files is shown in table 9.2.

Table 9.2 The Java classes generated from the CreditCardPayment WSDL using wsimport

Generated class	Description
`CreditCardPayment.java`	The JAX-WS service endpoint interface. It can be used as a remotable interface for SCA Java component services or references.

`AddressType.java`
`CreditCardDetailsType.java`
`CreditCardTypeType.java`
`PayerType.java`
	The JAXB classes to represent the complex types defined by the XML schema. For example, `CreditCardDetailsType` represents the detail information about the credit card.

Table 9.2 The Java classes generated from the CreditCardPayment WSDL using wsimport *(continued)*

Generated class	Description
`AuthorizeType.java` `AuthorizeResponseType.java`	Wrapper types that hold the parameters passing in and out of the authorize operations. These are required to support wrapper-style encoding of parameters. You can read the details of wrapper-style encoding in section 2.3.1.2 of the JAX-WS specification.
`ObjectFactory.java` `package-info.java`	The JAXB object factory to create JAXB objects and the package-info file containing the XML schema target namespace for the given package.

The wsimport tool will generate a Java interface for the service that the WSDL describes, as shown here.

Listing 9.3 AX-WS generated interface for the CreditCardPayment component

```
package com.tuscanyscatours.payment.creditcard;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService (name = "CreditCardPayment",
targetNamespace = "http://www.example.org/CreditCardPayment/")
@XmlSeeAlso({
    ObjectFactory.class
})
public interface CreditCardPayment {

    @WebMethod (action =
          "http://www.example.org/CreditCardPayment/authorize")
    @WebResult (name = "Status",
          targetNamespace = "")
    @RequestWrapper(localName = "authorize",
          targetNamespace = "http://www.example.org/CreditCardPayment/",
          className = "org.example.creditcardpayment.AuthorizeType")
    @ResponseWrapper(localName = "authorizeResponse",
          targetNamespace = "http://www.example.org/CreditCardPayment/",
          className ="org.example.creditcardpayment.AuthorizeResponseType")
    public String authorize(
                      @WebParam (name = "CreditCard", targetNamespace = "")
                      CreditCardDetailsType creditCard,
                      @WebParam(name = "Amount", targetNamespace = "")
                      float amount);
}
```

This generated Java interface looks complicated because it's annotated automatically with JAX-WS annotations to capture additional metadata based on the corresponding

WSDL and XSD. For example, `@WebParm` customizes the mapping of an individual parameter to a WSDL message part and XML element. `@WebResult` customizes the mapping for the return value.

If a plain Java interface with no annotations is written and used as the service interface, then the Tuscany runtime applies the default mapping rules defined by JAX-WS when handling maps between the service interface and the SOAP envelopes on the wire. For example, the `targetNamespace` will be derived from the Java package name and the parameter names will be assumed to be `arg0`, `arg1`, `argN`, and so on. For more information on the default mapping, please refer to chapter 3 of the JAX-WS specification.

Bob adopts the interface that's generated by the JAX-WS wsimport tool. He also uses the generated JAXB classes that appear as parameters to the operations in the generated interface to represent the data in his Java component implementation. For example, the following listing shows the generated JAXB class used to represent the CreditCardDetails structure that he must pass to the authorize operation of the CreditCardPayment component.

Listing 9.4 JAXB generated class for the CreditCardDetails data structure

```
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.XmlAccessType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "CreditCardDetailsType", propOrder = {
    "creditCardType",
    "creditCardNumber",
    "expMonth",
    "expYear",
    "cardOwner",
    "cvv2"
})
public class CreditCardDetailsType {

    @XmlElement(name = "CreditCardType")
    protected CreditCardTypeType creditCardType;

    @XmlElement(name = "CreditCardNumber")
    protected String creditCardNumber;

    @XmlElement(name = "ExpMonth")
    protected Integer expMonth;

    @XmlElement(name = "ExpYear")
    protected Integer expYear;

    @XmlElement(name = "CardOwner")
    protected PayerType cardOwner;

    @XmlElement(name = "CVV2")
    protected String cvv2;

}
```

1 Name and order of child elements

2 Denotes an XML element

We've omitted the setter and getter methods from this class definition for readability. In this JAXB class, annotations are placed in the Java code by the wsimport tool to accurately record the mapping from the Java class to the XML schema type definition that was used to generate it. For example, `@XmlType` (1) provides the name and the ordering of child elements. `@XmlElement` (2) denotes a local XML element and specifies its name. This approach works well if you start from WSDL and XSD. If you start from written Java objects, then JAXB has a set of default mapping rules to map Java types into XML types.

Bob uses the generated `CreditCardDetailsType` type explicitly when creating his component implementation. The following listing shows how this generated JAXB type can be used.

Listing 9.5 Using generated JAXB classes within a component implementation

```
import org.osoa.sca.annotations.Reference;

public class PaymentImpl implements Payment {

    @Reference
    protected CreditCardPayment creditCardPayment;

    public String makePaymentMember(String customerId,
                                    float amount) {

        CustomerDetails customerDetails = getCustomerDetails(customerId);

        ObjectFactory objectFactory = new ObjectFactory();
        CreditCardDetailsType ccDetails =
                              objectFactory.createCreditCardDetailsType();

        ccDetails.setCreditCardType(CreditCardTypeType.fromValue("Visa"));

           PayerType ccOwner = objectFactory.createPayerType();
        ccOwner.setName(customerDetails.getCustomerName());
        ccDetails.setCardOwner(ccOwner);
        String status = creditCardPayment.authorize(ccDetails,
                                                    amount);
       }

    public String makePaymentNonMember(CreditCardDetailsType ccDetails,
                                       float amount) {
        String status = creditCardPayment.authorize(ccDetails,
                                                    amount);
       }
}
```

More business logic goes here

In this example the Payment component service defines two methods to show two ways in which data can be dealt with. In the `makePaymentMember` operation, it's assumed that the customer has preregistered with TuscanySCATours and provided their credit card details. The details are retrieved from the database, and the `CreditCardDetailsType` object is created field by field to pass to the CreditCardPayment

component. This is a good example of the component implementation performing some mediation between distinct application-level data types.

In the `makePaymentNonMember` operation, the client application collects the credit card details and passes them in. Bob passes the object straight through his logic and on to the CreditCardPayment component with no further intervention.

Now you've seen how Bob makes a request to the CreditCardPayment component. Let's look at how Mary implements the service side.

9.2.2 Accepting data in component services as SDO objects

Mary takes a different approach than Bob. There are many different options for implementing the CreditCardPayment component, such as these:

- A legacy application that's wrapped as a web service
- A web service implemented using C++ or .NET
- A web service implemented in Java

From the data-representation perspective, the CreditCardDetails data can also be represented in many different ways. The key is that Mary's decision can be completely independent of Bob's, although what she's implementing will be called by Bob's Payment component.

Mary also decides to use Java to implement the CreditCardPayment component and will expose it as a web service using `binding.ws`. But instead of JAXB she chooses to use Service Data Objects (SDO) to represent the data. SDO is a data programming architecture and API. The Tuscany project provides an implementation of SDO, and the SCA runtime supports SDO as a databinding.

The main purpose of SDO is to simplify data programming so that developers can focus on business logic instead of the underlying technology used to represent data. SDO provides uniform access to data that can reside in heterogeneous sources such as XML, RDB, Java, and SOAP. SDO provides other benefits such as change history and disconnected model. We won't cover all that here because this is a book about SCA. If you'd like to learn more about SDO, the Tuscany project and SDO specification that are published on OSOA and OASIS are a good starting point.

Let's focus on the implementation details related to using SDO to represent data. Like JAX-WS and JAXB, SDO allows you to generate Java interfaces and classes from XML schema. As an alternative to statically generated Java classes that represent data structures, SDO also provides a dynamic API (`commonj.sdo.DataObject`) so that you can directly work with XML data in Java without code generation. In this next example we'll use the static approach. The steps are similar to those used by Bob where he generated JAXB classes from WSDL for the client side.

Mary first generates the SDO classes from the CreditCardPayment.wsdl file. This can be done using the SDO static code generator, which is a command-line tool in Tuscany for generating Java source code (static SDOs) for data types defined in an XML

schema. The generator is used as follows (assuming the SDO tools are available on the classpath):

```
java org.apache.tuscany.sdo.generate.XSD2JavaGenerator -targetDirectory
     target/sdo-source -javaPackage payment.creditcard -prefix
     CreditCardPayment -noNotification -noContainment -noUnsettable src/main/
     resources/wsdl/ CreditCardPayment.wsdl
```

The structure of the SDO project code is described on the Apache Tuscany website (https://cwiki.apache.org/TUSCANY/sdo-project-code-structure.html). This includes details of the command-line arguments for this SDOstatic code generator.

As an alternative to the command-line generator, you can use the SDO Maven plug-in to generate the Java classes. We use Maven in the TuscanySCATours example, and the following is included in the creditcard-payment-sdo contribution pom.xml file:

```
<plugin>
  <groupId>org.apache.tuscany.sdo</groupId>
  <artifactId>tuscany-sdo-plugin</artifactId>
  <version>1.1.1</version>
  <executions>
    <execution>
      <id>generate-sdo</id>
      <phase>generate-sources</phase>
      <configuration>
      <schemaFile>
         ${basedir}/src/main/resources/CreditCardPayment.wsdl
      </schemaFile>
      <javaPackage> com.tuscanyscatours.payment.creditcard</javaPackage>
      <prefix>CreditCardPayment</prefix>
      <noNotification>true</noNotification>
      <noContainer>true</noContainer>
      <noUnsettable>true</noUnsettable>
      </configuration>
        <goals>
          <goal>generate</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

Table 9.3 shows the set of interfaces and classes that result from running the SDO static code generator.

Table 9.3 SDO classes generated from the CreditCardPayment WSDL document

Generated class	Description
AddressType.java AuthorizeResponseType.java AuthorizeType.java CreditCardDetailsType.java PayerType.java	The Java interfaces for the XML data types

Table 9.3 SDO classes generated from the CreditCardPayment WSDL document *(continued)*

Generated class	Description

`AddressTypeImpl.java`
`AuthorizeResponseTypeImpl.java`
`AuthorizeTypeImpl.java`
`CreditCardDetailsTypeImpl.java`
`PayerTypeImpl.java`
	The implementation classes of the Java interfaces

`CreditCardPaymentFactory.java`	The factory interface that can be used to create SDO objects
`CreditCardPaymentFactoryImpl.java`	The implementation class of the factory

Let's look at the SDO version of the `CreditCardDetailsType`. SDO generates an interface as well as an implementation class. The interface doesn't have any SDO dependencies, and so it can potentially be implemented using other technologies. The following listing shows the generated `CreditCardDetailsType`. The comments that appear in this generated code have been omitted for readability.

Listing 9.6 Generated SDO interface for `CreditCardDetailsType`

```
package com.tuscanyscatours.payment.creditcard;

import java.io.Serializable;

public interface CreditCardDetailsType extends Serializable {
    String getCreditCardType();
    void setCreditCardType(String value);
    String getCreditCardNumber();
    void setCreditCardNumber(String value);
    int getExpMonth();
    void setExpMonth(int value);
    int getExpYear();
    void setExpYear(int value);
    PayerType getCardOwner();
    void setCardOwner(PayerType value);
    String getCVV2();
    void setCVV2(String value);
}
```

There is a generated factory interface and class called `CreditCardPaymentFactory`. It can be used to create instances of `CreditCardDetailsType`.

Using these generated data types, Mary can now manually define a remotable service interface as follows:

```
@Remotable
public interface CreditCardPayment {

    public String authorize(CreditCardDetailsType creditCard,
                            float amount);
}
```

Based on this remotable service interface, Mary can go ahead and create an implementation for the service. The implementation class will implement the interface.

```
@Service(CreditCardPayment.class)
public class CreditCardPaymentWSImpl implements CreditCardPayment {

    public String authorize(CreditCardDetailsType creditCard,
                            float amount) {
        String cardNumber = creditCard.getCreditCardNumber();
        PayerType owner = creditCard.getCardOwner();
        ...
    }
}
```

As the last step, the CreditCardPayment component service can be exposed using the Web Services binding by configuring the component in the following way:

```
<component name="CreditCardPayment">
  <implementation.java
  class="com.tuscanyscatours.payment.creditcard.impl.CreditCardPaymentImpl"/>
  <service name="CreditCardPayment">
   <interface.wsdl interface= "http://www.example.org/CreditCardPayment/#
                                          wsdl.porttype(CreditCardPayment)"
   <binding.ws/>
  </service>
</component>
```

We've walked through the steps that Bob and Mary take to develop the two SCA components that exchange data over the Web Services protocol. Bob and Mary independently choose to use two different databindings (JAXB versus SDO) in their business logic to represent the same business data. Their choices are independent of the data-representation technologies imposed by the protocol stack, in this case, Web Services SOAP over HTTP.

Although it's unlikely that you'll be implementing the same application as Bob and Mary did, you'll face the same problems of how to describe data structures on a service interface and how to create suitable data objects so that you can manipulate data in your component implementations. You may also be faced with a situation where different databindings are required to represent the same information. The techniques that are shown here apply whatever the specific nature of your application or the data you have to deal with.

Having said this, we'll continue to follow Bob and Mary as they look into how the decisions they've made so far about using WSDL, JAXB, and SDO can be reflected in SCA component definitions.

9.3 Describing data contracts within SCA compositions

An SCA component has multiple places where you can specify the interfaces for SCA services and references. Figure 9.4 shows that component types, references, services, and bindings can all be configured with interface information.

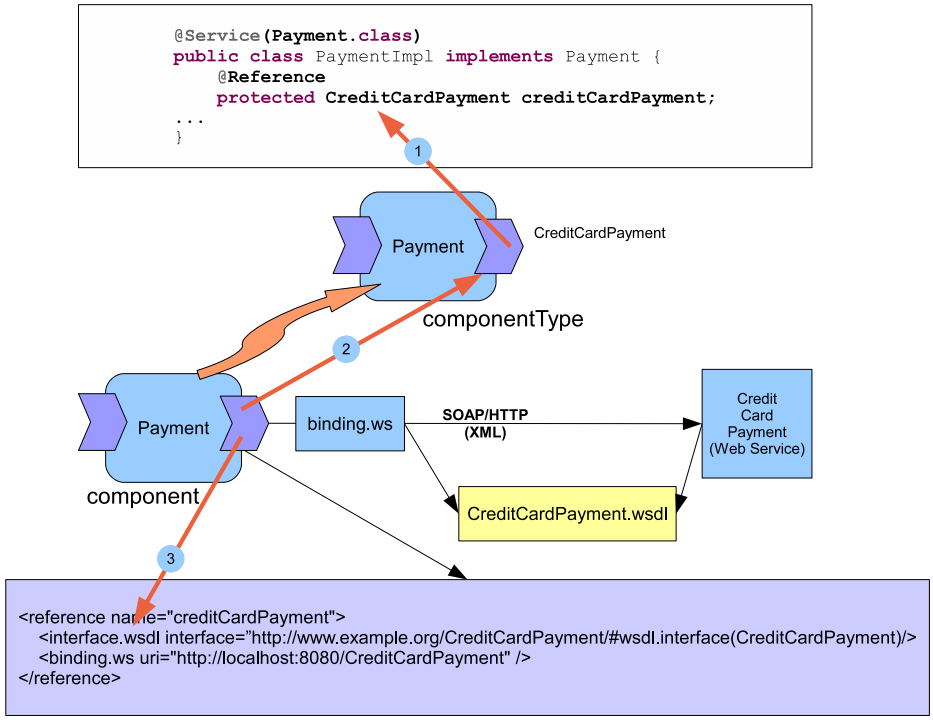


Figure 9.4 Interface information can be configured through reference and service interfaces in the component type. It can also be configured through interface elements in the reference and service elements of a component in the composite file. Interface configuration can also be imposed by some bindings.

Arrow 1 in figure 9.4 shows that a Java field declaration annotated with `@Reference` in the implementation class is introspected as a reference of the component type. Section 9.3.1 describes how interface configuration is determined for a component type. Arrow 2 indicates that the component reference is inherited from the component type.

The component reference can be further configured with `interface.wsdl` and `binding.ws`, as illustrated by arrow 3. Section 9.3.2 discusses the impact of configuring the reference with an `interface.wsdl` element. Section 9.3.3 goes on to discuss the impact of binding specific interface configuration.

These same configuration points are also applicable to component services. This section gives you enough information to determine where interface configuration is required in your application.

9.3.1 Specifying contracts on the component type

The Payment component owned by Bob is a Java component. It configures the `creditCardPayment` reference with a Web Services binding.

```
<component name="Payment">
<implementation.java
```

```
    class="com.tuscanyscatours.payment.impl.PaymentImpl" />
     <reference name="creditCardPayment">
         <binding.ws uri="http://localhost:8080/CreditCardPayment" />
     </reference>
</component>
```

What interface is used for the `creditCardPayment` reference? The SCA assembly specification says that if the interface for a component reference isn't explicitly configured in the composite file, then it inherits its interface from the component type. The component type represents the contract of the component implementation and can be derived in one of two ways: Either it can be specified explicitly in a component-type side file, or, for most implementation types, Java included, the component type can be derived by the runtime by inspecting the Java implementation class. From the data-handling perspective, the data types imposed by the implementation class dictate the representations of data that the component will consume and produce. The following listing shows an outline of the Payment component implementation. This will be used when determining the component type by introspection.

Listing 9.7 Introspecting the component type from the component implementation

```
@Service(Payment.class)
public class PaymentImpl implements Payment {

    @Reference
    protected CreditCardPayment creditCardPayment;

    public String makePaymentMember(String customerId,
                                    float amount) {
    }

    public String makePaymentNonMember(CreditCardDetailsType ccDetails,
                                       float amount) {
    }
}
```

1 Component reference

2 Component service operations

Business logic removed for readability

2

Business logic removed for readability

In this case Tuscany builds the component type by introspecting the component implementation class, `PaymentImpl`, and the interfaces it implements. In particular, the runtime can detect that this component has a reference, `creditCardComponent` 1, which uses the `CreditCardPayment` interface. It can also detect that the component exposes a Payment service with two operations, `makePaymentMember` and `makePaymentNonMember` 2. Please refer to the description of implementation.java in chapter 5 to understand the rules that map Java members into SCA component type constructs.

Note that the creditCardPayment field is annotated with `@Reference`. It defines an SCA reference named `creditCardPayment`. The interface of the reference is `com.tuscanyscatours.payment.creditcard.CreditCardPayment`. The component implementation code is written to talk to the CreditCardPayment component through this Java interface. The Java types for the parameters and return value denote the data representation that the component uses to communicate with the

service provider. In this case we know those types are JAXB objects because we've previously been through the steps involved in building this interface.

The data types for the service that the component exposes can be found by looking at the parameters that the two operations require. In this case it's a combination of Java primitive types and the `CreditCardDetailsType` JAXB type.

9.3.2 Specifying contracts on component services and references

What if the interface is reconfigured at the component level? For example, the following code shows how an `<interface.wsdl>` element can be added to the composite description:

```
<component name="Payment">
  <implementation.java
   class="com.tuscanyscatours.payment.creditcard.CreditCardPaymentImpl" />
  <reference name="creditCardPayment">
    <interface.wsdl interface="http://www.example.org/CreditCardPayment/#
                               ➥ wsdl.interface(CreditCardPayment)"/>
    <binding.ws uri="http://localhost:8080/CreditCardPayment" />
  </reference>
</component>
```

Now the component reference is further configured with a WSDL interface. Interestingly though, this won't change how the Payment component implementation code works with data. The Payment component talks to the CreditCardPayment component using the `CreditCardPayment` interface, which is the Java type for the field annotated with `@Reference`.

The interface for the component reference is used to constrain the algorithm that wires SCA references to SCA services. For example, the WSDL can be tailored to match only a subset of available component services. The `<interface.wsdl/>` element also provides information for the binding, for example, the WSDL required to define the web service.

Equally, a component's services can be configured in the same way. In this case the interface description serves to restrict the service interface that's exposed through the service binding. The interface configured in the service element must be a compatible subset of the interface offered by the service implementation.

9.3.3 Providing contract configuration to bindings

The binding configuration for a reference might impact how the data is represented by the binding extension code. In our case, `<binding.ws>` can point to a WSDL element, and the WSDL will be used to define the wire format for the SOAP message. For example, if `<binding.ws>` is configured as follows, the Web Services binding uses the WSDL service CreditCardPaymentService and WSDL port CreditCardPaymentPort as configuration when sending and receiving SOAP envelopes:

```
<binding.ws wsdlElement="http://www.example.org/CreditCardPayment/#wsdl.
     port(CreditCardPaymentService/CreditCardPaymentPort)"/>
```

This ability to configure bindings is important because the details of a communication protocol used by a service may be exacting. For example, in the case of Web Services, the SCA reference may be required to send SOAP messages using document literal encoding or other specific forms of encoding. This detailed configuration can be achieved by providing the Web Services binding with a WSDL interface configured in precisely the right way.

This binding-specific configuration is also applicable to bindings that appear within component service elements of a composite file.

You've now seen how interface configuration information can be applied in various places in our SCA application. From this you understand how a component's services and references are typed either through explicit configuration of the component type or composite files or through introspection of the component implementation itself. We'll now look at how data is transformed in order that it will be presented in the right format in references and services, in the bindings they use, and on the wire between references and services.

When business data is exchanged over various protocols, different wire formats can be used to encode the data. For example, web services use XML messages, JSON-RPC uses JSON, and RMI-IIOP uses serialized Java objects. Different implementations of the same protocol stacks may choose to use different data representations. For example, in the Web Service domain, some examples are as follows:

- Axis1 uses DOM.
- Axis2 uses AXIOM.
- JAX-WS uses JAXB.

You've already seen that when data is received from the bits and bytes of the transport, many different data representations are available in the Java programming language, for example:

- Raw data, such as `byte[]`, `String`, `InputStream`, or `Reader`
- XML parsing technologies, such as DOM, SAX events, or StAX
- Java/XML binding technologies, such as JavaBeans, JAXB, SDO (generated static SDO classes or dynamic DataObject), or AXIOM

All of these options mean that the Tuscany runtime must be flexible enough to provide suitable transformations as data flows from reference to service and back again. Section 9.4 gives an overview of various transformations that are required in order that data be in the right format at every point between reference and service.

9.4 Data transformations

In an SOA environment it's not always possible to enforce a unified data representation technology for all service communications. Relying on business logic to convert data from one form to another removes some of SCA's promised flexibility because it couples the service implementation, service collaboration, and service communication

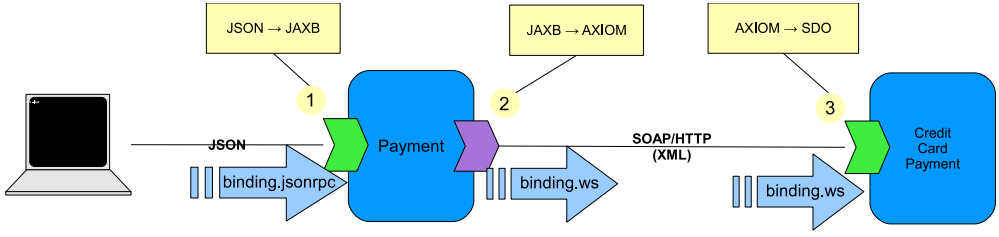


Figure 9.5 Component interaction and data flow for the credit card payment sample

technologies tightly together. To exchange data in different representations without the intervention of business logic developers, we'll need to introduce data transformation capabilities.

Bob and Mary have chosen to represent the data in different ways for their collaborating services. Data needs to be transformed so that the components can handle the data in their own formats. Let's first understand what data transformation would be required for a round trip between the Payment and CreditCardPayment components from the TuscanySCATours scenario.

To purchase a travel package, the customer selects credit card as the method of payment. The customer isn't a registered user, so when it comes time to pay for their chosen trip, they fill out a form on the web page to provide information such as credit card number, expiration date, holder's name, and billing address. When the customer clicks the confirm button, the web browser sends the request to the Payment component. The Payment component then calls the external CreditCardPayment component over Web Services to authorize the charge. The status of the transaction is updated on the web page so that the customer knows whether it goes through.

Figure 9.5 shows how the various stages of this process take place and how data must be transformed between components.

The following sections describe the requirements for each of the three main transformations in turn. Each section refers to the corresponding stage number in figure 9.5. The process starts in the web browser JavaScript, which takes the credit card data from the HTML form and populates a JavaScript object. It then makes an Ajax invocation to the Payment component running on the server using the JSON-RPC protocol. In the browser the data is represented as a JavaScript object and exchanged over the HTTP connection using JSON format.

9.4.1 Converting the data coming from the browser from JSON to JAXB

This is shown as stage 1 in figure 9.5. The Payment component is implemented by a Java class. It provides a service that takes the credit card information as a JAXB object. The service is configured with `binding.jsonrpc` to service the Ajax requests from the web browser. The binding receives the JSON data from the JSON-RPC layer, and it needs to be converted into the JAXB object so that the Java implementation for the Payment component can handle it.

9.4.2 Converting from JAXB to AXIOM in order to send a SOAP request

This is shown as stage 2 in figure 9.5. The Payment component has a reference to the CreditCardPayment component using the Web Services protocol in order to authorize the credit card charge. The Java implementation makes the web service request using a Java interface, which passes a JAXB object. The JAXB object has to be transformed into an AXIOM object so that the Axis2 stack can marshal it into a SOAP message. Web service responses are unmarshalled back into JAXB from the AXIOM/XML representation provided by Axis2.

9.4.3 Converting from AXIOM to SDO

This is shown as stage 3 in figure 9.5. The Web Services binding for the CreditCardPayment component receives the credit card transaction from the Axis2 stack as an AXIOM object, and it needs to be converted into SDO so that CreditCardPayment can handle it. Responses coming back from the CreditCardPayment component are transformed from SDO back into AXIOM ready for the Web Services binding to create the response SOAP envelope.

These data transformations seem to be complicated and they're technology oriented. If the application developers were forced to manually convert the data from one format to the other, it would be a burden. The business logic would become brittle again because it would be tied to a set of data representation technologies that are peculiar to the binding technology. We would also lose the flexibility to connect references and services that represent the same data in different ways. But don't worry, because you'll now see how Tuscany helps Bob and Mary by introducing a databinding framework that provides transparent data transformation between components without the intervention of a component's business logic.

9.5 The Tuscany databinding framework

In order to facilitate data handling in the SOA environment, the Tuscany SCA infrastructure provides a databinding framework that handles data conversions automatically at points in the architecture where data conversion is required. This preserves application flexibility and simplifies the development of bindings. Figure 9.6 shows

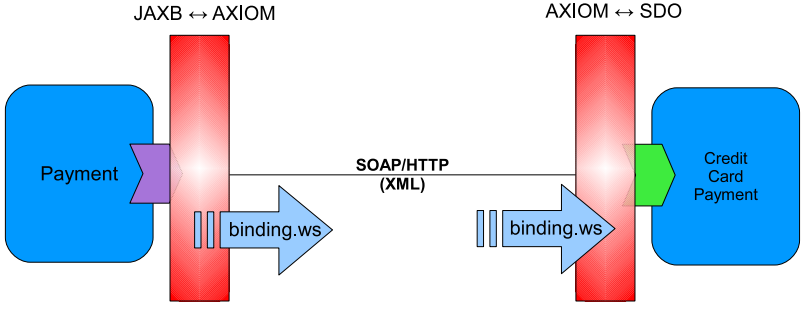


Figure 9.6 Automatic data transformation between Payment and CreditCardPayment components

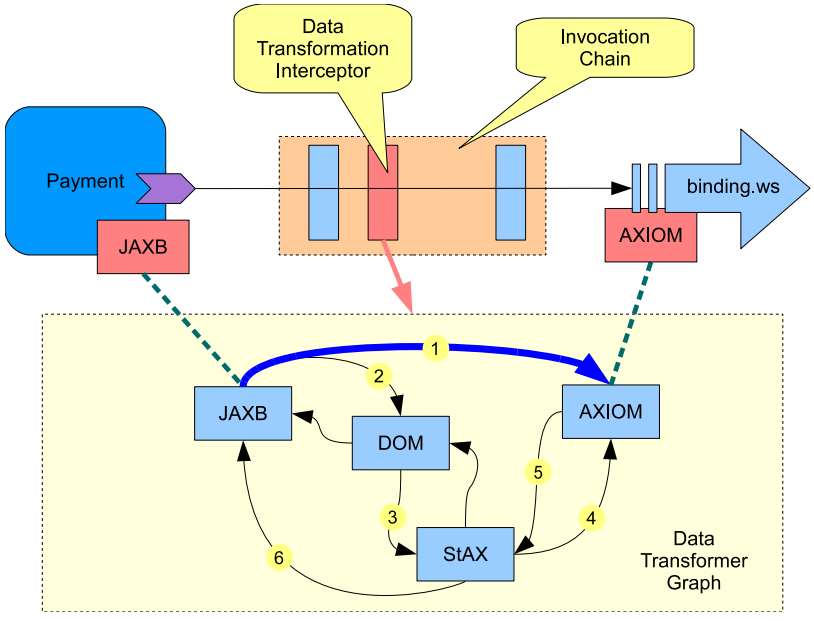


Figure 9.7 The Tuscany databinding framework uses a graph of transformers (1–6) to find a path (1) to transform data from JAXB to AXIOM for the Payment component reference using `binding.ws`.

where the Tuscany runtime transparently transforms the data passing between the Payment and CreditCardPayment components without intervention from the business logic.

Tuscany supports popular databindings such as JAXB, SDO, DOM, StAX, and AXIOM. It also provides the ability to extend the architecture to easily add new ones. The Tuscany databinding framework is illustrated in figure 9.7, transforming data at a component reference. The same process happens for a component service.

There are a few primary functions in the databinding framework. Databinding abstracts the different representations of data. Code to transform from one data representation to another is plugged into the Tuscany runtime as databinding transformers. Transformers (marked as 1–6 in figure 9.7) connect databindings, and they form a graph for the possible paths by which data can be transformed. Figure 9.7 shows a path with transformer 1 chosen to convert data from JAXB to AXIOM.

Component implementation extensions use the Tuscany extension programming interfaces to provide information about which data representations can be handled by the business logic. In some cases the details of the expected data type can be specified by the component implementation itself via the component service interface. You've seen how this is achieved in our CreditCardPayment example, where the Java interface describes operations with parameters using specific data representation technologies. In our example, the data is represented using JAXB and SDO.

Binding extensions use the Tuscany SPI to tell the Tuscany runtime which data representations are expected by the protocol stack. The data can then be converted from the application code into a supported representation by the binding and vice versa. The Tuscany runtime will establish the following paths for data transformation.

- SCA component reference > SCA reference binding
- SCA service binding > SCA component service

If there is a mismatch between the data representation used by the component and the data representation used by the binding, a data transformation interceptor will be added to perform the data transformation. This transformation is transparent to application developers. When transformation is needed, the interceptor finds a path in the transformer graph and invokes the transformers on the path to convert the data from the source representation to the target representation. There may be multiple paths between two databindings. If a direct transformer is available, the Tuscany runtime will use it; otherwise, it'll find the most efficient path between the two databindings. In the case that no path can be found, a system exception will be thrown to indicate that the two components can't interact using the data formats as is.

9.6 Summary

In this chapter, we've shown the different aspects of data handling in an SCA environment through the sample credit card payment scenario.

As you learned previously in this book, with SCA and Tuscany it's straightforward to reconfigure and assemble a business solution, for example, you can

- Change the implementation technology
- Change the reference to service wiring
- Change the communication protocol

As you've seen from Bob and Mary's experience in this chapter, in order to exploit this flexibility, application developers need the freedom to choose their preferred data representation. Components with compatible data but with incompatible data representations, such as JAXB and SDO, are able to interoperate without changing the business logic. You've seen that data representation can be considered in three different places:

- Component implementation
- Component references and services and their bindings
- Communication protocols between bindings

The same SCA service can be implemented using different technologies that support or require different data representations. Some implementation types such as implementation.java don't mandate how data will be represented, and it's up to component developers to choose the appropriate format. Other implementation types, such as implementation.bpel, require specific data representations, in this case XML. It all depends on what data format the component implementation engine supports.

In SCA, service providers are decoupled from service consumers. Collaborating services don't have to adopt the same data representations for the same data structure. It isn't even feasible in some cases; for example, the service consumer and service provider may be owned by two different organizations, which have their own preferred data representations. The same SCA service can be consumed by more than one service that provides input data and handles output data with different representations. So the data representation for a service provider is agnostic in relation to the service consumers.

SCA also separates the communication protocols from the business logic. Declarative bindings make it possible to switch protocols without changing the component implementation code. Furthermore, the same SCA service and references can be configured with different bindings so that they can communicate over different protocols that require different wire formats. For example, the Payment component can be configured using both `binding.ws` and `binding.jsonrpc` so that it can be exposed as a Web Service and a JSON-RPC service. The Web Services protocol uses an XML-based wire format, whereas JSON-RPC uses JSON. Similar requirements apply to SCA references too.

Tuscany provides a framework that simplifies the data representation and transformation so that business application developers can have the freedom to choose their preferred data representation technology. This allows developers to implement business logic without worrying about complex data transformations between references and services. This decoupling of data representation allows SCA to deliver on its promises of flexibility and agility. A developer can rewire a component and even change the binding technology without having to change the component implementation to take account of differing data representations.

Now that you understand how data can be represented and transformed, it's time to take a wider view of an SCA application and look at how statements of policy can be applied, which are independent of component implementations, their reference, services, and bindings, and, indeed, the data that passes between them.

10 Defining and applying policy

This chapter covers

- SCA policy in general
- Implementation policy
- Interaction policy

Policies in Tuscany and SCA are used to control those aspects of your application that tend to be orthogonal to your component implementations, for example, logging and monitoring or security concerns such as authentication, integrity, and confidentiality. These types of functions are often referred to as qualities of service and can significantly complicate an application if you implement them inside your business logic.

In an enterprise, it's better to define policies once and then apply them consistently across all of the components that need them. That's why SCA describes a policy framework that allows intents and policy sets to be defined and associated with component implementations and the bindings that connect them. Using policy sets and intents, you don't have to use APIs to achieve quality of service (QoS) behavior; you just mark up the composite application.

You've already learned in previous chapters that components built using Tuscany and SCA are independent of some of the complexities you normally associate with enterprise application development. You've seen how bindings can be added to component services and references without changing the component implementation itself. Tuscany and SCA treat the configuration of policies in the same way by allowing you to define and configure policies independently of the implementations and bindings to which they're attached.

We'll start our description of policy in SCA and Tuscany with an overview of how policy fits into the SCA domain concept. We'll then look at how the Tuscany runtime applies policy configuration to the running application, where the rubber hits the road, as it were.

To bring policy into focus we'll look at how an implementation policy, which enables message logging, can be applied to an SCA component implementation. We'll also look at an interaction policy that enables simple authorization between the Payment and CreditCardPayment components from the travel sample. The interaction sample takes up quite a bit of space in this chapter because it shows how intents and policy sets are applied to both the reference and service sides of the interaction. We'll finish the chapter with some discussion of the more advanced features of the policy framework and a quick review of the limited policies that Tuscany currently supports.

10.1 An overview of policy within an SCA domain

The SCA policy framework is defined in the SCA Policy Framework Specification (http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf). The foundations of SCA's approach to supporting policy are the policy intents and policy sets that SCA defines.

A policy intent, as you might have guessed by the name, is a statement of intent. For example, say you want to have all of the messages that arrive at a component logged for debugging purposes. You could associate the intent logging with your component implementation. This doesn't say how logging will be performed. It just says that you want logging performed.

To implement the logging feature you'll have to define a policy set that satisfies the logging intent. A policy set describes the technical details involved in logging messages, for example, what details to log and where the log messages should be written.

The approach of having policy intents defined separately from policy sets allows a great deal of flexibility when constructing and deploying SCA applications. The component developer can use intents to describe QoS requirements abstractly without needing to understand the mechanisms used to provide the QoS features. At the deployment stage, enterprise policy sets ensure that intents are satisfied in accordance with local policy, for example, by using whatever technical infrastructure is most appropriate in the local enterprise environment.

All SCA intents and policy sets are provided in configuration files with the name definitions.xml. There can be multiple definitions.xml files spread across application

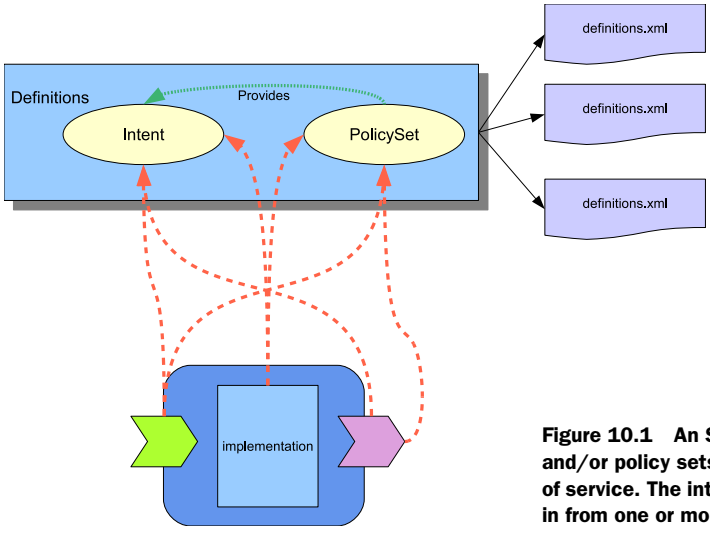


Figure 10.1 An SCA component with intents and/or policy sets attached to enforce quality of service. The intents and policy sets are read in from one or more definitions.xml files.

contributions and Tuscany extensions. Figure 10.1 shows how policy sets and intents within one or more definitions.xml files can be associated with component implementations, references, and services.

As illustrated in the figure, SCA provides the ability to describe intents and policy sets in an SCA domain. The SCA domain isn't shown explicitly, but all of the items shown have been contributed to a single SCA domain. The figure shows that intents and policy sets can be associated with implementations, what SCA not surprisingly calls *implementation policies*. This allows you to control the quality of service provided by the component implementation either at the level of the whole implementation or at the level of individual operations. In our logging example, every message entering or leaving the implementation, or operation, can be treated consistently in terms of what information is recorded about the message.

Figure 10.1 also shows that intents and policy sets can be associated with component references and services, what SCA calls *interaction policies*. This allows policy to be associated with the individual bindings that are configured for component services and references. In this way the protocol stack of a binding can be configured to ensure that appropriate QoS behavior is achieved.

As can be seen from the figure, intents and policy sets are described inside XML files with the name definitions.xml. A definitions.xml file can either appear inside a contribution or be packaged with a Tuscany extension such as an implementation or binding type. The latter approach is used in those cases where the intents or policy sets are provided as a feature of the extension in question. For example, the Tuscany `binding.ws` extension ships with a definitions.xml file that includes intents for selecting SOAP versions called SOAP.1_1 and SOAP.1_2.

The contents of all of the definitions.xml files present in a domain, whether they come from extensions or from contributions, are aggregated and made available

across the domain. When configuring an application there's no need to worry about the precise location of a definitions.xml file, just that it's available within the domain.

With this brief overview, we can look at how to use intents and policy sets to enable QoS features. We'll start at the end by describing what impact policy has on the Tuscany runtime before looking at how to use intents and policies.

10.2 The policy runtime

We have quite a few ways of configuring policy in an SCA composite application. The flexibility provided is useful, but it does make the policy framework seem a little complicated at first sight. The objective, though, is quite straightforward. Our aim is to add Java code to bindings or component implementations in order to enable QoS features.

Before we look in more detail at the policy model, this section takes a bottom-up approach by looking at how Tuscany allows policy code to be added to the runtime in the form of policy interceptors. This should answer any nagging doubt about how a policy configuration leads to something actually happening in the running application. With this knowledge, some of the mystery will be removed, and in section 10.3 we'll continue to look at the various ways policy intents and policy sets can be used.

10.2.1 Policy interceptors

The Tuscany framework is designed to allow policy-related Java code to be added easily. Between each binding and implementation, on both service and reference ends of an SCA wire, the Tuscany framework creates chains of interceptors, one for each service operation. Each interceptor in the chain operates on incoming and outgoing messages. The processing performed depends on the interceptor in question; for example, a databinding interceptor might transform the contents of the message from one data format to another. A policy interceptor, on the other hand, might encrypt or decrypt a message.

Figure 10.2 gives an abstract view of some of the runtime artifacts that are created by Tuscany when you create a wire between two SCA components.

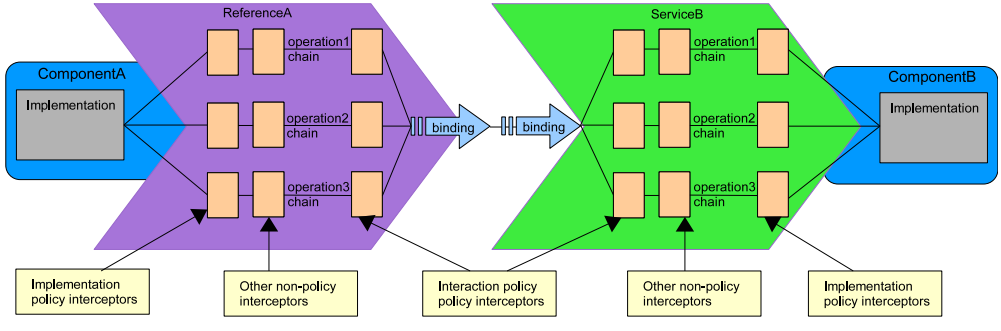


Figure 10.2 A high-level view of how the Tuscany runtime uses message interceptors strung out along a chain to allow policy function to be plugged in. There's an interceptor chain for each operation within each component service and reference.

In figure 10.2, ComponentA has a reference called ReferenceA, which is wired to ServiceB of ComponentB. The reference and service have a compatible binding, the type of which isn't specified here because this mechanism is the same regardless of which binding you choose.

Within the reference and service are three rows, or chains, of interceptors. A chain is built for each operation that the interface of ReferenceA and ServiceB provides. In this case the interface has three operations: operation1, operation2, and operation3.

A message for operation2 sent from ComponentA, via ReferenceA, passes along the middle chain of three interceptors. First, it's processed by the implementation policy interceptors. Next, it's processed by any other interceptors that happen to be on the chain; for example, interceptors are used here to perform databinding transformations. Finally, the message is processed by the interaction policy interceptors before being passed on to the reference binding that transports the message, using some protocol, to the service binding.

At the service side the process is reversed. The message passes from the binding into the middle of the three chains that were constructed to process messages for operation2. First, the message passes through the interaction policy interceptor before carrying on through any other interceptors on the chain. Finally, it's processed by the implementation policy interceptor and is passed on to operation2 in the implementation of ComponentB.

With this chain of interceptors, it's not hard to see how policy functions such as logging or encryption could be implemented and added to a running application using a Tuscany interceptor.

10.2.2 The interceptor interface

Each interceptor on the chain has to be able to do two things. It has to process a message and then pass the message on to the next interceptor. To achieve this, each interceptor implements a simple Tuscany-provided interface called `Interceptor`.

```
public interface Interceptor extends Invoker {
    void setNext(Invoker next);
    Invoker getNext();
}
```

You can see that the `Interceptor` interface allows interceptors to be linked together into a chain of interceptors. The Tuscany framework adds the next interceptor in the chain by calling `setNext()` on the last interceptor in the chain. The `Invoker` base class provides the interface that passes the message along the chain and is equally straightforward:

```
public interface Invoker {
    Message invoke(Message msg);
}
```

The `Message` type here is a Tuscany interface that represents messages in a general way as they pass between the binding and the component implementation. We'll provide a more general discussion of these two interfaces, and interceptor chains, in chapter 13.

A simple example of an interceptor is the logging implementation policy interceptor we've already mentioned. The following code shows a simplified version of the Tuscany logging policy interceptor that uses JDK logging to print out information about messages that are passing into or out of component implementations.

Listing 10.1 A simplified version of the `JDKLoggingPolicyInterceptor`

```
public class JDKLoggingPolicyInterceptor implements Interceptor {
    private Invoker next;

    public JDKLoggingPolicyInterceptor(String context,
                                       Operation operation,
                                       PolicySet policySet) {
    }

    public Message invoke(Message msg) {
        logger.logp(Level.INFO,
                    context,
                    "",
                    "Invoking operation - " + operation.getName());
        return getNext().invoke(msg);
    }

    public Invoker getNext() {
        return next;
    }

    public void setNext(Invoker next) {
        this.next = next;
    }
}
```

Configuration mostly omitted

Call the next interceptor

In this much-reduced version of the `JDKLoggingPolicyInterceptor`, all of the setup code has been omitted, and the `invoke` method prints out only the name of the operation and no details of the message that's passing through. When the interceptor is added as an implementation interceptor at reference or service ends of the wire, the operation name is printed for each message that's sent. You can see the full implementation in the Tuscany source policy-logging module.

The purpose of showing you this isn't to provide an in-depth tutorial of policy interceptor construction but to show you that, at the end of the day, a policy interceptor is acting on messages passing between bindings and component implementations.

Now that you know a bit about how policy is implemented using an interceptor, let's look at some examples of how a policy interceptor is configured using policy intents and policy sets.

10.3 Using intents and policy sets for implementation policy

In this first example we'll use the Trip component from the travel-booking application and look at how to configure the logging implementation policy. You can find this example in the Tuscany SCA Tours trip-policy contribution. You can run it with the

policy launcher from launchers/policy. This launcher pulls in the following contributions as well: common, payment-java-policy, and policy-client. Figure 10.3 shows what the Trip component looks like.

The Trip component in the `trip-policy` contribution provides two services, Search and Book, which allow you to search for prepackaged trips and then book one. Because we're going to configure an implementation policy in this case, the policy will apply to the component implementation regardless of which service is being called. Let's look at the composite description first.

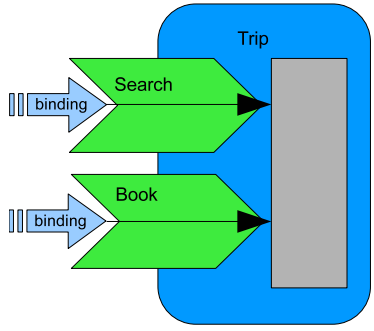


Figure 10.3 The Trip component providing Search and Book services. Messages flow from the service bindings to the implementation, as shown by the solid lines with arrows.

10.3.1 Adding implementation intents to the composite file

Let's start by adding a policy intent to the description of the Trip component that appears in a trip.composite file:

```
<component name="TripComponent">
    <implementation.java class="com.tuscanyscatours.trip.impl.TripImpl"
                         requires="tuscany:logging"/>
    <service name="Search"/>
    <service name="Book"/>
</component>
```

The important piece of configuration here, as far as policy is concerned, is the attribute on the implementation.java element, which reads

```
requires="tuscany:logging"
```

This tells the Tuscany runtime that the intention is to log messages passing into and out of the implementation. You may be wondering how we know that that's what this means. The string `logging` isn't all that descriptive. As you learned in the overview, each intent is described in a definitions.xml file. In this case the definitions.xml is provided by the Tuscany developers inside the Tuscany source policy-logging extension module and is as follows:

```
<definitions xmlns="http://www.osoa.org/xmlns/sca/1.0"
             targetNamespace="http://tuscany.apache.org/xmlns/sca/1.0"
             xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
             xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0">

    <intent name="logging"
             constrains="sca:implementation.java
                         sca:implementation.spring">
        <description>
            All messages to and from this implementation must be logged
        </description>
```

```
    </intent>
</definitions>
```

First, note that intents aren't intrinsically meaningful. The human-readable description here is the only thing that tells us what this intent means.

Second, note that the intent specifically describes what sort of elements in the composite file it can be applied to. In this case the `constrains` attribute says that the `logging` intent can be used with both implementation.java and implementation.spring.

It's worth pointing out here that the `requires` attribute that allows you to add intents to the SCA composite application can be added to pretty much any element in the composite file. The `constrains` attribute doesn't stop you from adding intents in the wrong place; it just tells the Tuscany runtime in which cases it should take notice of the intents. It's also the case that child elements in the composite file inherit intents that are specified on their parent elements; we'll cover more on this later in the chapter.

You've seen how the intent is defined. Let's now look at how the runtime interprets the intent and how it chooses a policy set that satisfies the intent.

10.3.2 Choosing a policy set to satisfy the intent

Intents don't do much in isolation. The Tuscany runtime needs to find a policy set that satisfies the intent in order to achieve the effect that's intended. In this example the intention is to log any message passing into and out of an implementation, so we'll need a policy set that tells the Tuscany runtime to do this.

Policy sets are specified in definitions.xml files. In this case a definitions.xml file is provided with the TuscanySCATours trip-policy contribution with the following contents:

```
<definitions xmlns="http://www.osoa.org/xmlns/sca/1.0"
             targetNamespace="http://tuscany.apache.org/xmlns/sca/1.0"
             xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
             xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
             xmlns:scatours="http://scatours">

    <policySet name="JDKLoggingPolicy"
               provides="tuscany:logging"
               appliesTo="sca:implementation.java">
        <tuscany:jdkLogger name="myLogger">
            <logLevel>FINER</logLevel>
        </tuscany:jdkLogger>
    </policySet>
</definitions>
```

This describes a policy set called JDKLoggingPolicy. The job of this policy set is to configure the Tuscany runtime to apply the implementation logging policy to the implementations of individual components.

How does Tuscany know which component implementations to apply this policy to? You can see that this policy set has an attribute called `provides`. This attribute describes which intents are satisfied by this policy set. In this case it's the `tuscany:logging` intent we've already described. There's also an `appliesTo` attribute, which

tells the runtime which bindings or implementation types this policy set can work with. In this case this policy set is configured to work only with implementation.java. Given these two pieces of information, the Tuscany runtime can find the component implementations that have the `logging` intent applied to them, and from these, which ones this policy set is able to work with.

The policy configuration information is provided inside the policy set. Several types of information can appear here, as described later in this chapter, but for now you can see the `tuscany:jdkLogger` element. The Tuscany policy-logging extension reads this element and creates an internal model to represent the JDK logging policy. It's configured by the `logLevel` element, which describes how much detail should be logged.

At runtime, Tuscany uses the internal model of the JDK logging policy to create a `JDKLoggingPolicyInterceptor` for the Trip component implementation. `JDKLoggingPolicyInterceptor` is included in the Tuscany distribution in the policy-logging module. The result is shown in figure 10.4.

The result of the policy configuration, through intents and policy sets, is the addition of the `JDKLoggingPolicyInterceptor` to incoming messages. We've already described how interceptors work. We're not showing the individual chains here, but this implementation policy interceptor will process all the messages coming into the implementation regardless of which operation of which service they're targeted at.

The operation of the `JDKLoggingPolicyInterceptor` causes log statements to be written out with varying levels of detail, depending on the `logLevel` configuration in the policy set. The operation of this policy, and the interceptor that does the real work, seems somewhat orthogonal to the real operation of the component implementation. This is intentional. The SCA Policy Framework is designed to allow these types of features to be added and configured in your composite application independently of the component implementations.

In this example we've focused on the Trip component, which doesn't define any explicit references. When references are defined by an implementation, any implementation intents and policy sets will be applied to the reference as well. If you want to control the way that policy interceptors are applied to individual services or references, you'll need to use an interaction policy.

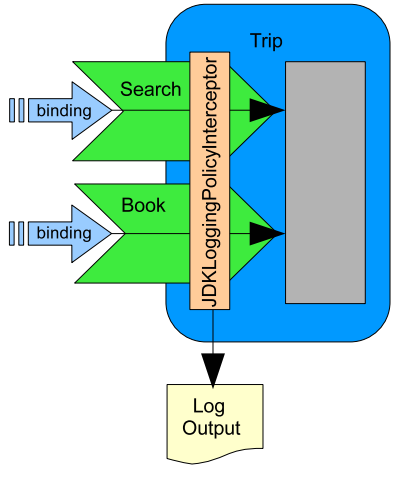


Figure 10.4 The result of applying the logging policy to the Trip component implementation is that the Tuscany runtime creates a `JDKLoggingPolicyInterceptor`. The chains aren't shown here, but the diagram indicates that the interceptor is applied to all operations of all services provided by the implementation of the Trip component.

10.4 Using intents and policy sets for interaction policy

Interaction policy is often the first type of policy we're asked about. This often happens because people new to SCA and Tuscany want to connect to or provide a service with some kind of security enabled. Interaction policy can be applied to both reference and service bindings. Regardless of whether the service or reference is connected to an SCA or non-SCA service, policy can be used to configure the operation of the service or reference's binding.

Each binding will interpret the policy configuration in a way that's appropriate to the protocol that the binding uses. For example, a binding that uses HTTP, such as the Web Services binding, may implement an authentication policy by passing basic authentication credentials in the HTTP header. Other bindings will likely use other authentication mechanisms.

To demonstrate how interaction intents and policy sets operate, we'll use the Web Services binding basic authentication example. We'll configure authentication between the TuscanySCATours Payment and CreditCardPayment components. These components are defined in the payment-java-policy and creditcard-payment-jaxb-policy contributions, respectively. Again, this example is run using the policy launcher. Figure 10.5 shows the message flow.

As with implementation policy intents, interaction policy intents can be added just about anywhere in the composite file. The SCA rules for the inheritance of policy information ensure that the Web Services binding is configured in the correct way.

This will be quite a long section because we'll use this Payment example to look at how interaction intents can be added into the application's composite file or into the service and reference definitions within a Java component implementation. We'll also look at how policy sets are defined to satisfy the intents on the service and reference sides of this example. We'll finish up by giving a brief overview of how the runtime is configured based on the intent and policy set provided. Let's start enabling authentication between the Payment and CreditCardPayment components by adding intents to the credit card composite file.

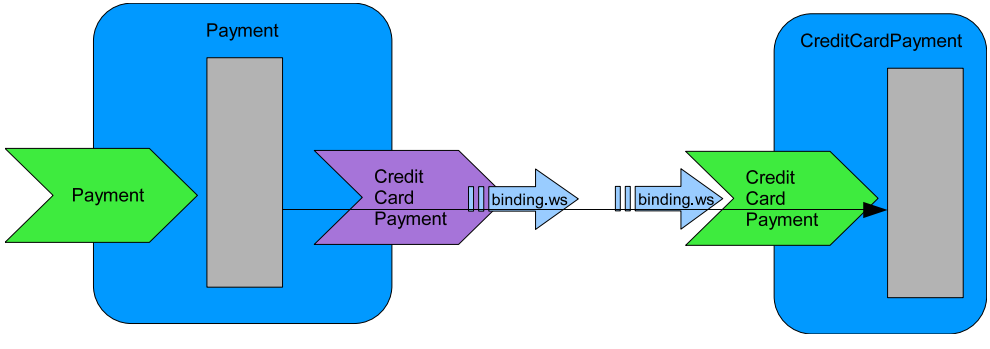


Figure 10.5 The Payment component is wired to the CreditCardPayment component, and the reference and service are configured to use the Web Services binding. Messages flow through the reference and service, as shown by the solid line with the arrow.

10.4.1 Adding interaction intents to the composite file

We'll start by configuring the service on the CreditCardPayment component with the `authentication` intent. The contents of the creditcard.composite file in the creditcard-payment-jaxb contribution that contains the CreditCardPayment component definition are shown in the following listing.

Listing 10.2 The credit card composite application

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
        targetNamespace="http://tuscanyscatours.com/"
        name="creditcard">

  <component name="CreditCardPayment">
    <implementation.java class="com.tuscanyscatours.payment.
    ➥ creditcard.impl.CreditCardPaymentImpl" />
    <service name="CreditCardPayment">
      <interface.wsdl interface="http://www.tuscanyscatours.com
      ➥ /CreditCardPayment/#wsdl.interface(CreditCardPayment)" />
      <binding.ws uri="http://localhost:8082/CreditCardPayment"
                  requires="authentication"/>                    (1) Authentication intent
      <binding.sca/>
    </service>
  </component>
</composite>
```

Listing 10.2 shows that the `authentication` intent has been added directly to the `<binding.ws/>` element using the `requires` attribute (1). This means that any message arriving at the CreditCardPayment service over the Web Services binding requires authentication.

Note that the default binding, `<binding.sca/>`, has no policy annotations, and so messages arriving over this binding don't require authentication. This sounds a little lax, so you could add authentication to the default binding in the same way it was added to the Web Services binding.

To make life a little easier and to prevent you from having to add intents on every binding, SCA allows intents and policy sets to be added at the service, component, and even the composite level. For example, if you want all bindings of the CreditCardPayment service to require that incoming messages are authenticated, you could write the following:

```
<service name="CreditCardPayment" requires="authentication">
  <interface.wsdl …/>
  <binding.ws uri="http://localhost:8082/CreditCardPayment"/>
  <binding.sca/>
</service>
```

This literally means that the service requires authentication and implies that all bindings of the service inherit the `authentication` intent. You can go further and say that

all services and references of a component require authentication. To do this you'd configure the `authentication` intent at the component level, as follows:

```
<component name="CreditCardPayment" requires="authentication">
  <service name="CreditCardPayment">
    <interface.wsdl …/>
    <binding.ws uri="http://localhost:8082/CreditCardPayment"/>
    <binding.sca/>
  </service>
</component>
```

If you want every service and reference of every component in your composite application to required authentication, then you'd move the `authentication` intent up one more level, as follows:

```
   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
              targetNamespace="http://tuscanyscatours.com/"
              name="creditcard"
              requires="authentication">
  <component name="CreditCardPayment">
    <service name="CreditCardPayment">
      <interface.wsdl …/>
      <binding.ws uri="http://localhost:8082/CreditCardPayment"/>
      <binding.sca/>
    </service>
  </component>
</composite>
```

In all of these examples, the Tuscany runtime ensures that the service and reference bindings are configured with the intents specified in parent elements. This mechanism applies to implementations as well, so when you configured the component and the composite in the last two snippets, the `authentication` intents would have potentially been added to the implementation as well.

But if you remember from section 10.3.1, intents are configured in the definitions.xml file with information about which elements they apply to. `Authentication` is an inbuilt intent that's defined in the following way in the Tuscany policy-security extension module's definitions.xml file:

```
    <intent name="authentication"
        constrains="sca:binding"/>
```

You can see that the authentication intent applies only to bindings. Even when this intent is defined at the component or composite level, it'll only be attached to service and reference bindings and won't be attached to implementations.

Defining intents using the `requires` attribute associated with elements in the composite file isn't the only way to add policy intents. Let's look next at how intents can be defined in a component implementation.

10.4.2 Adding interaction intents to the component implementation

In our payment example the Payment component's `creditCardPayment` reference also needs to be configured for authentication to match the expectations of the CreditCardPayment service you've configured. In this case you'll add the intent directly to the Payment component's Java implementation.

SCA provides the `@Requires` annotation to allow application developers to attach one or more intents directly to a Java implementation. See section 2 of the SCA Java Common Annotations and APIs specification for general information about SCA policy Java annotations (http://www.osoa.org/download/attachments/35/SCA_JavaAnnotationsAndAPIs_V100.pdf). Currently intents and policy sets can only be added directly to components with a Java implementation. Other implementation types aren't supported in the same way, and policy configuration for non-Java implementation types must be carried out by adding intents and policy sets to the composite file.

We're using a Java implementation in our example, and the code to add authentication to the creditCardPayment reference is shown here.

Listing 10.3 Defining intents in the Java implementation

```
@Service(Payment.class)
public class PaymentImpl implements Payment {

    @Reference
    protected CustomerRegistry customerRegistry;

    @Reference
    @Requires("{http://www.osoa.org/xmlns/sca/1.0}authentication")
    protected CreditCardPayment creditCardPayment;

    @Reference
    protected EmailGateway emailGateway;

    @Property
    protected float transactionFee = 0.01f;

    public String makePaymentMember(String customerId, float amount) {
        try {
            Customer customer = customerRegistry.getCustomer(customerId);
            String status =
                   creditCardPayment.authorize(customer.getCreditCard(),
                                               amount + transactionFee);
            emailGateway.sendEmail("order@tuscanyscatours.com",
                                   customer.getEmail(),
                                   "Status for your payment",
                                   customer + " >>> Status = " + status);
            return status;
        } catch (CustomerNotFoundException ex) {
            return "Payment failed due to " + ex.getMessage();
        } catch (AuthorizeFault_Exception e) {
```

The authentication intent (1)

```
            return e.getFaultInfo().getErrorCode();
        } catch (Throwable t) {
            return "Payment failed due to system error " + t.getMessage();
        }
    }
}
```

Note that the namespace-qualified `authentication` intent has been added directly to the `creditCardPayment` reference using the `@Requires` annotation ❶. Because there's no standard way of representing a QName as a string, SCA has adopted the syntax used by `javax.xml.namespace.QName`, where the namespace is declared inside curly braces in front of the qualified name.

The policy intent can be specified for various scopes, including Java interfaces, Java implementation classes, business methods, or fields. But method or field-level intents override class or interface-level intents. The intents defined via Java annotations can't be overridden by the same intents that might be defined later during assembly in the composite file. SCA follows the same inheritance rules to apply intent annotations as specified in section 2.1 of the common annotation specification, JSR 250 (http://jcp.org/en/jsr/detail?id=250).

If you look at the SCA Policy Framework specification, you'll also see that some specific annotations are described, such as `@Authentication`. Tuscany doesn't support these specific policy annotations yet and instead relies on the `@Requires` style, which achieves the same effect.

Adding authentication directly to the Payment component's implementation has the same effect as if we'd added the `authentication` intent to the component definition in the composite file in the following way:

```
<component name="Payment">
  <implementation.java class="com.tuscanyscatours.payment.impl.PaymentImpl"/>
  <service name="Payment">
    <binding.sca/>
    <binding.ws uri="http://localhost:8081/Payment"/>
  </service>
  <reference name="customerRegistry" target="CustomerRegistry"/>
  <reference name="creditCardPayment" requires="authentication">
    <binding.ws uri="http://localhost:8082/CreditCardPayment"/>
  </reference>
  <reference name="emailGateway" target="EmailGateway"/>
  <property name="transactionFee">0.02</property>
</component>
```

Note how `requires="authentication"` has been added to the `creditCardPayment` reference.

You may be wondering why SCA supports so many different ways of specifying policy configuration. It's all about flexibility and choice. If you're a component implementer who knows that a service will require authentication, then you can describe that intention by adding configuration to the implementation itself. The Tuscany runtime will then raise errors if this intent isn't satisfied at runtime.

If, on the other hand, you want to leave it up to the application assembler to configure whether each service must only accept authenticated messages, then you're free to do this.

We've made our intentions clear in our example now, but, as you know from section 10.3, when we discussed implementation policy, adding intents doesn't provide enough information to tell the Tuscany runtime how to implement the policy. For that we'll need policy sets.

10.4.3 Choosing a policy set to satisfy the intent at the service

As we described in section 10.3, policy sets tell the Tuscany runtime how to realize the intents that are attached to the composite application. In our authentication example we're going to exploit the basic authentication policy defined in Tuscany, which can be applied to the Web Services binding. You'll define a policy set in your CreditCardPayment module's definitions.xml file from the creditcard-payment-jaxb-policy contribution, as shown here.

Listing 10.4 The CreditCardPayment definitions.xml file

```
<definitions xmlns="http://www.osoa.org/xmlns/sca/1.0"
             targetNamespace="http://itest/policy"
             xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
             xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0">

    <policySet name="BasicAuthenticationPolicySet"
               provides="authentication"
               appliesTo="sca:binding.ws">
        <tuscany:basicAuthentication>
          <tuscany:userName>myname</tuscany:userName>
          <tuscany:password>mypassword</tuscany:password>
        </tuscany:basicAuthentication>
    </policySet>
</definitions>
```

1 Intent it's associated with

2 Elements it operates on

3 The policy itself

First, you'll define the policy set and give it the name `BasicAuthenticationPolicySet`. The attributes of the policy set say that it provides policy for the `authentication` intent (1) and that it can be applied only to the SCA Web Services binding (2). This restriction is applied because what follows causes the Tuscany runtime to run code that will work only with the Web Services binding.

If you had attached the `authentication` intent to the CreditCardPayment component, then, as previously described, the result would be that the `authentication` intent would be added to the SCA default binding as well as the Web Services binding. In this case, the Web Services binding authentication would be handled by the `BasicAuthenticationPolicySet` shown here. The authentication function of the default binding would not be handled by this policy set because of the `appliesTo` restriction (2); that is, it doesn't apply to `binding.sca`. You'd have to define an extra policy set that tells the Tuscany runtime how to handle authentication over the default SCA binding.

Looking inside the policy set you see the `<tuscany:basicAuthentication>` element (3). By their nature SCA policy sets are extensible, and this basic authentication element is a Tuscany-specific element that configures the Web Services binding for basic authentication.

On the service side the presence of this type of policy causes a basic authentication message interceptor to be introduced and also causes the Web Services binding to extract the basic authentication credentials from the incoming message and make them available for the interceptor to process. In the default implementation that Tuscany ships with, all that the interceptor does is compare the incoming credentials with the username and password specified in the policy set. The following listing shows the code for the `BasicAuthenticationServicePolicyInterceptor`'s `invoke` method. This can be found in the Tuscany policy-security module.

Listing 10.5 `BasicAuthenticationServicePolicyInterceptor invoke`

```
public Message invoke(Message msg) {
  Subject subject = SecurityUtil.getSubject(msg);
  BasicAuthenticationPrincipal principal =
    SecurityUtil.getPrincipal(subject,
                              BasicAuthenticationPrincipal.class);
    boolean authenticated = false;

    if (principal != null){

      System.out.println("Authenticating user: " +
                         principal.getName());

      if (policySet != null) {
        for (Object policyObject : policySet.getPolicies()){
          if (policyObject instanceof BasicAuthenticationPolicy){
            BasicAuthenticationPolicy policy =
                                (BasicAuthenticationPolicy)policyObject;
            if (policy.getUserName().equals(principal.getName())){
              if (policy.getPassword().equals(principal.getPassword())){
                authenticated = true;
              }
            }
          }
        }

      }
    }

    if (authenticated == false){
      throw new ServiceRuntimeException("User: " +
                                        principal.getName() +
                                        " cannot be authenticated");
    }

    return getNext().invoke(msg);
}
```

The `invoke` method retrieves the policy subject from the incoming message, retrieves the principal from the subject, and then compares the name and password in the principal with the name and password provided in the policy set.

This policy isn't exactly what you'd call a production-quality policy and is severely restricted in the amount of flexibility that it allows for managing authenticated users. But it does demonstrate the approach, and it would be easy to extend or replace the interceptor in order to call out to some more proficient security provider such as an LDAP-based registry.

For our example to work properly, you'll need to have a policy set that will provide an implementation for the `authentication` intent that's attached to the `creditCardPayment` reference of the Payment component.

10.4.4 Choosing a policy set to satisfy the intent at the reference

In this example you're running the Payment and CreditCardPayment components in separate composites on separate nodes. This means that you'll need to define a separate definitions.xml file for the Payment component. This is because the two composites are effectively running in separate domains. But the policy set is identical to the policy set we provided for the CreditCardComponent.

```
<policySet name="BasicAuthenticationPolicySet"
                provides="authentication"
                appliesTo="sca:binding.ws">
    <tuscany:basicAuthentication>
      <tuscany:userName>myname</tuscany:userName>
      <tuscany:password>mypassword</tuscany:password>
    </tuscany:basicAuthentication>
</policySet>
```

At the reference side this causes the Tuscany runtime to add an interceptor to the message flow for the `creditCardPayment` reference. This interceptor takes the username and password from the policy set and adds it to the message passing along the chain. The presence of this policy set also configures the Web Services binding to take the credentials from the message and add them to the HTTP header of the outgoing message. The following snippet shows that the `BasicAuthenticationReferencePolicyInterceptor` `invoke` method is simplistic:

```
public Message invoke(Message msg) {
  Subject subject = SecurityUtil.getSubject(msg);
  BasicAuthenticationPrincipal principal =
    SecurityUtil.getPrincipal(subject,
                              BasicAuthenticationPrincipal.class);

  if (principal == null &&
      policy.getUserName() != null &&
      !policy.getUserName().equals("")) {
    principal = new BasicAuthenticationPrincipal(policy.getUserName(),
                                                 policy.getPassword());
    subject.getPrincipals().add(principal);
  }
```

```
  return getNext().invoke(msg);
}
```

The code first looks to see if any principal information has been placed on the message by the implementation. This could be the case when the implementation is operating within an existing security context. Assuming that no principal information is found, a principal is constructed based on the information contained in the policy set.

The reference side of the basic authentication policy is more useful than the service side in real applications because it allows you to talk to non-SCA web services that exploit basic authentication.

10.4.5 Running the payment example with authentication enabled

You've added the `authentication` intent at both the `CreditCardPayment` reference and service, and you've added policy sets to describe what the Tuscany runtime should do to implement these intents. Now you can run the sample, using the payment-java-policy launcher, and see the effect of this configuration.

In this case the launcher runs two nodes in effectively separate domains. The first node is configured with the creditcard-payment-jaxb-policy contribution that contains the following files:

```
creditcard-payment-jaxb-policy/
    META-INF/
        sca-contribution.xml
    com.tuscanyscatours.payment.creditcard/
        CreditCardPayment.java
    com.tuscanyscatours.payment.creditcard.impl/
        CreditCardPaymentImpl.java
    creditcard.composite
    CreditCardPayment.wsdl
    definitions.xml
```

The layout and contents of this contribution should be familiar to you by now. The sca-contribution.xml file describes the contribution, and the creditcard.composite file describes the composite application, which, in this case, has the CreditCardPayment component in it. The Java and WSDL files describe the component service interface and the component implementation. The important file that's packaged with this contribution as far as policy is concerned is the definitions.xml file that describes the basic authentication policy we've already looked at.

The second node starts two contributions: the policy-client contribution and the payment-java-policy contribution. The payment-client contribution adds a client component to call the Payment component so that it in turn will call the CreditCardPayment component. Of most interest is the Payment contribution, which contains the following files:

```
payment-java-policy/
    META-INF/
        sca-contribution.xml
```

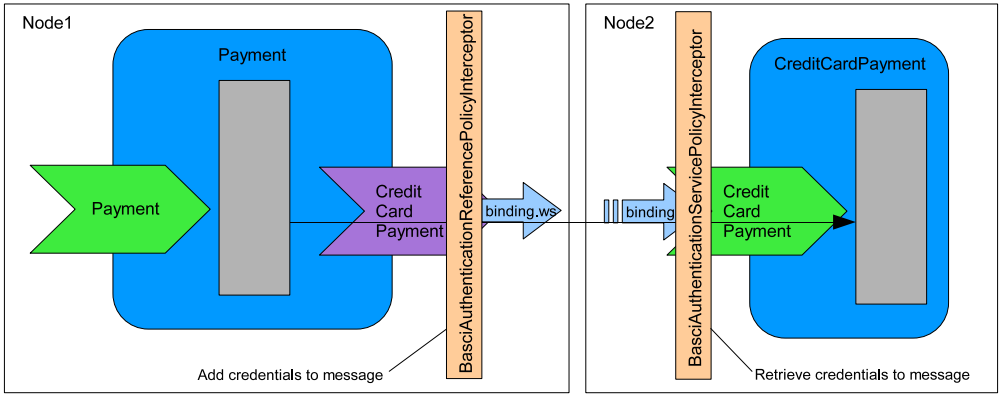


Figure 10.6 The result of applying the authentication policy to the Payment and CreditCardPayment components is that the Tuscany runtime creates a basic authentication interceptor in both reference and service chains and configures the Web Services binding to use the HTTP header to pass basic authentication credentials.

```
com.tuscanyscatours.payment/
    Payment.java
com.tuscanyscatours.payment.impl/
    PaymentImpl.java
payment.composite
Payment.wsdl
CreditCardPayment.wsdl
definitions.xml
```

Again, everything is pretty much as you'd expect to find it (we've omitted some of the Java files here to keep the listing short), but notice that there's another definitions.xml file. This again holds the definition of the basic authentication, but this time the policy will be applied to the reference.

Figure 10.6 shows the components and the effects of the policy configuration in enabling message authentication between the Payment and CreditCardPayment components.

If you were to look at the HTTP message that passed between the reference and service Web Services bindings in this case, you'd see something like the message shown in the following listing.

Listing 10.6 The credit card authorization message with basic authentication enabled

```
POST /CreditCardPayment HTTP/1.1
Content-Type: text/xml; charset=UTF-8
SOAPAction: "http://www.tuscanyscatours.com/CreditCardPayment/authorize"
User-Agent: Axis2
Authorization: Basic bXluYW1lOm15cGFzc3dvcmQ=
Host: localhost:8082
Content-Length: 504

<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
```

HTTP basic authentication credentials (1)

```
            xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns2:authorize
              xmlns:ns2="http://www.tuscanyscatours.com/CreditCardPayment/">
      <CreditCard>
        <CreditCardType>Visa</CreditCardType>
        <CreditCardNumber>1111-2222-3333-4444</CreditCardNumber>
        <ExpMonth>1</ExpMonth>
        <ExpYear>2012</ExpYear>
        <CardOwner>
          <Name>John Smith</Name>
        </CardOwner>
        <CVV2>1234</CVV2>
      </CreditCard>
      <Amount>100.02</Amount>
    </ns2:authorize>
  </soapenv:Body>
</soapenv:Envelope>
```

The important line is the line in the HTTP header that starts with `Authorization` (1), which has been added by virtue of the intent and policy set configuration. We should point out that you wouldn't want to send credit card details without encrypting them using just basic authentication. You'd want to have the content encrypted using a confidentiality intent. But this is just a sample so that you can see the effect of adding intents.

Now if you were to disable the reference-side policy by removing the `authentication` intent from the reference, for example, the sample would fail with an error because the service side still expects messages to contain credentials.

We've spent a lot of time looking at how to add intents and then define policy sets for interaction policies, in this section, and implementation policies, in the previous section. Such policies allow you to configure the QoS provided by the component services you build. Interaction policy also allows you to configure QoS when communicating with other SCA or non-SCA services.

Now that you have the basics, let's take a quick look at some of the other features that the policy framework specification includes.

10.5 Other features of the SCA Policy Framework

Now that we've walked through how intents and policy sets are used in general, you should be aware of a few other features.

We'll look at some of the flexibility you have if you're using intents. Profile intents can be used to group other intents together. Qualified intents can be used to structure your intent's namespace, and finally the definitions.xml file can be used to define the default intents provided by SCA bindings.

We'll start by looking at how policy sets can be attached directly to the composite application without first associating intents with the composite application. This is useful if the person assembling the application wants to be explicit about which policy sets should be used.

10.5.1 Dealing with policy sets directly

In the previous sections, where we discussed implementation and interaction policy, we started by adding intents to the composite application and then defined policy sets that were linked to the abstract intents using the `provides` attribute. This use of abstract intents is useful when you want to separate the role of the person who sets the QoS intention from that of the person who configures policy implementation in your particular environment. But the use of intents isn't mandatory, and you can use policy sets directly in both the composite file and in Java implementations.

In the composite file a policy set is attached directly to an element using the `policySets` attribute. In our previous payment example we could have configured the `CreditCardPayment` reference in the composite file in the following way:

```
<component name="Payment">
  <implementation.java
     class="com.tuscanyscatours.payment.impl.PaymentImpl" />
  <service name="Payment">
    <binding.sca/>
    <binding.ws uri="http://localhost:8081/Payment"/>
  </service>
  <reference name="customerRegistry" target="CustomerRegistry"/>
  <reference name="creditCardPayment"
             policySets="BasicAuthenticationPolicySet">
    <binding.ws uri="http://localhost:8082/CreditCardPayment"/>
  </reference>
  <reference name="emailGateway" target="EmailGateway"/>
  <property name="transactionFee">0.02</property>
</component>
```

Note that the `creditCardPayment` reference now has a `policySets` attribute, which references the BasicAuthenticationPolicySet policy set directly.

This equivalent approach can be taken in this component's Java implementation. SCA provides the `@PolicySets` annotation to allow Java developers to specify policy sets. A policy name is a string in the form of a qualified name (QName). As we said previously, SCA has adopted the syntax used by `javax.xml.namespace.QName` where the namespace is declared inside curly braces in front of the qualified name, for example:

```
@PolicySets("{http://www.osoa.org/xmlns/sca/
     1.0}BasicAuthenticationPolicySet")
```

`@PolicySets` can be specified for two or more policy set names using a comma-separated list, for example:

```
@PolicySets({"{http://www.osoa.org/xmlns/sca/1.0}FirstPolicySet",
             "{http://www.osoa.org/xmlns/sca/1.0}SecondPolicySet"})
```

`@PolicySets` can be specified for various scopes in the implementation including Java interfaces, Java classes, methods, or fields. The method or field-level policy sets override the class or interface ones. SCA follows the same inheritance rules to apply policy set annotations as specified by the Common Annotations for Java Specification,

JSR 250. Policy sets must satisfy intents expressed for the implementation when both are present.

10.5.2 Profile intents

Intents can be used to collect other intents. This makes it easier for the assembler to configure combinations of intents. For example, if you wanted to configure a reference or service with a combination of intents that include integrity and confidentiality, you could define a profile intent as follows:

```
<sca:intent name="messageProtection"
      constrains="sca:binding"
      requires="confidentiality integrity">
      <sca:description>
         Protect messages from unauthorized reading or modification
      </sca:description>
 </sca:intent>
```

When a profile intent is used, the Tuscany runtime will automatically replace the profile intent with the individual intents that it requires, and processing will continue, as we've already described.

10.5.3 Intent qualification

The SCA Policy Framework Specification provides a way to group intents, where an intent can be further qualified to add more detail to the description of the QoS that's required. A good example of this is the `SOAP` intent that can be used to control the version of SOAP that the Web Services binding uses. The SCA Web Services Binding Specification defines the following intents:

```
<sca:intent name="SOAP" constrains="sca:binding.ws">
    <sca:description>
        Communication through this binding requires SOAP
    </sca:description>
</sca:intent>

<sca:intent name="SOAP.1_1">
    <sca:description>
        Communication through this binding requires SOAP 1.1
    </sca:description>
</sca:intent>

<sca:intent name="SOAP.1_2">
    <sca:description>
        Communication through this binding requires SOAP 1.2
    </sca:description>
</sca:intent>
```

This simplifies the definition of the group of intents. The main or qualifiable intent, `SOAP`, defines what elements this intent constrains. The qualified intents, `SOAP.1_1` and `SOAP.1_2`, serve to extend the `SOAP` intent to provide further fidelity of configuration.

10.5.4 Default intents

A definitions.xml file can contain information about what intents an extension supports by default. The `<bindingType>` and `<implementationType>` elements support this configuration. For example, the Web Services binding, which is described in chapter 7, has a definitions.xml file with the following configuration:

```
<sca:bindingType type="sca:binding.ws"
                 mayProvide="SOAP SOAP.1_1 SOAP.1_2 MTOM"
                 alwaysProvides=""/>
```

This says that the `binding.ws` extension can optionally be configured with the `SOAP`, `SOAP.1_1`, `SOAP.1_2`, or `MTOM` intents, and it'll know what to do with them without your having to provide a policy set. These intents are configured as `mayProvide` because they can optionally be attached to `binding.ws` in the composite file.

If there are intents that an extension provides regardless of how the composite application is configured, they'd appear in the `alwaysProvides` attribute.

We've covered a good deal of ground now and explained how you can attach policy to various parts of your composite application. Let's now take a short tour of the policies that Tuscany currently supports.

10.6 Tuscany intents and policy sets

Intents and policies in Tuscany have been developed as required by the Tuscany community. There aren't that many so far, and it's no surprise that intents and policies have been created first for popular bindings, such as the Web Services binding.

Table 10.1 provides an index of some of the more useful intents and policies that Tuscany currently supports. To understand how to use these in detail, look at the examples referenced in the description text.

In general, the intents will already be defined in definitions.xml files shipped with Tuscany extensions, such as `binding.ws`. To satisfy the intents, you'll need to build a policy set inside a definitions.xml file included in one of your application's contributions. Table 10.1 lists the policy elements you can use to build a policy set. Where the table says that the policy is provided by an extension, this means that it's a default intent and there's no need to define a policy set.

Table 10.1 Summary of useful intents and policies in Tuscany

Intent	Policy	Description
`binding.ws`		
`SOAP`	Provided by `binding.ws`	Qualified interaction intent that can be used to configure the Web Services binding to use either the SOAP1_1 or SOAP1_2 protocol.
`MTOM`	Provided by `binding.ws`	Interaction intent that tells the Web Services binding to pass binary data using the Axis2 MTOM support.

Table 10.1 Summary of useful intents and policies in Tuscany *(continued)*

Intent	Policy	Description
authentication	<tuscany: basicAuthentication/>	Interaction intent/policy that tells the Web Services binding to use HTTP-based basic authentication. See payment-java-policy and creditcard-payment-jaxb-policy.
authentication integrity confidentiality	<tuscany:wsConfigParam>	Interaction intent/policy that can be used to set Axis2 configuration parameters. See sample/helloworld-ws-reference-secure, where it's used to set Inflow and OutflowSecurity parameters.
authentication integrity confidentiality	<wsp:Policy/>	Interaction intent/policy that uses WS policy to configure Axis Rampart. See sample/helloworld-ws-reference-secure.
binding.jms		
priority	<tuscany:jmsHeader/>	Interaction intent/policy used to demonstrate the use of policy as an alternative to more direct configuration of the binding.
deliveryMode	<tuscany:jmsHeader/>	Interaction intent/policy used to demonstrate the use of policy as an alternative to more direct configuration of the binding.
binding.http		
authentication	<tuscany: authenticationConfiguration/>	Interaction intent/policy used to configure an HTTP endpoint to request expect HTTP basic authentication. See samples/store-secure.
confidentiality	<tuscany:confidentiality>	Interaction intent/policy used to configure an HTTP endpoint to use an HTTP transport. See samples/store-secure.
General		
tuscany:logging	<tuscany:jdkLogger>	Logs messages passing into or out of an implementation.

If the policy is an interaction policy such as authorization or integrity, the Tuscany runtime will apply it before the implementation is called. Therefore, interaction policies are invisible to the implementation. Any processing required by the policy is performed by the Tuscany runtime within the reference or service code without implementation involvement.

At the time of writing, only Java implementations support the inclusion of annotations for policy intents and policy sets. You can't associate intents and policy sets directly

with Spring applications, BPEL processes, or scripts. You can, though, associate implementation intents and policies with the implementation element in the composite file.

Because we have relatively few policies implemented at present, as described in table 10.1, it's likely that the precise QoS configuration you require isn't already supported, and so we encourage you to look at the existing policy extensions and improve them for your own needs and even better consider contributing any improvements to the Tuscany project.

10.7 Summary

Supporting quality of service in an application is one of those things that we all know we have to do. We also know that having to code this kind of functionality in each component implementation we build doesn't lead to happiness. We complicate our business logic and repeat code that others are probably better qualified to construct.

The Tuscany and SCA Policy Framework provides a structured way of defining and building the QoS functions required by our SCA applications. The Policy Framework allows us to do this independently of the component implementations we build. This has a number of important benefits. Our business logic isn't mixed up with QoS code. The QoS features can be developed at an enterprise level and can be applied consistently across many components and applications. Finally, it supports the clear separation of roles between the business logic developers and those responsible for assembling and deploying SCA applications in the enterprise environment.

You've now had good look at the policy framework and the ways in which you can use intents and policy sets to configure your applications. This brings this section on using the detailed features of Apache Tuscany to a close. We're going to move on now and look at how to deploy the applications you build.

Part 3

Deploying Tuscany applications

Part 3 describes some of the different environments in which Tuscany can run, and demonstrates the full Tuscany SCA Tours travel-booking application. In chapter 11, "Running and embedding Tuscany," you'll learn about some of the different ways that Tuscany can operate. You'll first look at how Tuscany can be run standalone to create applications. Following that, you'll see how Tuscany can be integrated into web containers such as Tomcat and WebSphere and application containers such as Java EE and OSGi.

In chapter 12, "A complete SCA application," the SCA concepts that we've discussed throughout the previous chapters are brought together to demonstrate the full TuscanySCATours travel-booking application. You'll learn how the application can be run in a single Java VM and how to distribute the same application across multiple Java VMs without changing the application code. The chapter wraps up with some hints and tips for developing SCA applications.

11 Running and embedding Tuscany

This chapter covers

- Understanding basics of the Tuscany runtime environment
- Running Tuscany in a standalone, web, or distributed environment
- Embedding Tuscany within a managed application container

Tuscany can be hosted in many different environments. It can be run standalone or invoked programmatically as a library. Tuscany is integrated with web containers such as Tomcat, Jetty, and WebSphere to enable web applications with SCA. Tuscany can also be embedded within other application containers such as Java EE or OSGi-based containers. When running Tuscany standalone, you use the Tuscany launcher to run an SCA application from the command line or within an IDE. In the other cases, the host environment manages the Tuscany runtime and SCA applications.

All of these options give you the flexibility to run your SCA applications in the environment of your choice. You aren't restricted to a single way of starting the

runtime. If you find that you need to embed the runtime to suit your particular needs, then Tuscany provides a set of APIs to allow you to do this.

In this chapter we'll walk through the various options that Tuscany provides for running SCA applications. Some of the options have been used in previous chapters. You can find various samples in the following locations:

- launchers/*
- contributions/creditcard-payment-webapp

Let's start by looking at the nodes and the SCA domain within which they run.

11.1 Understanding the Tuscany runtime environment

In chapter 3 you learned how to run an SCA composite application in a local domain with a single execution node or as a distributed domain with multiple execution nodes. Before we start to explore different options to host Tuscany, you'll first need to understand the key actors, such as the SCA domain and the Tuscany node, that make up the runtime environment for SCA applications. We'll explain what the domain and node are, what information needs to be configured for a node, and how the hosting environment controls the node and interacts with other nodes in the same SCA domain.

11.1.1 The SCA domain and Tuscany nodes

An SCA domain manages a set of composite applications that can be connected using SCA wires and constrained by the same policy definitions. An SCA domain consists of the following:

- A virtual domain-level composite (that is, a domain composite) whose components are deployed and running
- A set of installed contributions that contain composites, implementations, interfaces, and other artifacts

An SCA domain represents a complete configuration of the SCA assembly. The components within an SCA domain can be hosted by one or more runtime environments that provide the necessary technology infrastructure. Tuscany introduces the node concept to represent a group of components (typically declared in the same composite) that can be started, executed, and stopped together on a Tuscany runtime. Figure 11.1 illustrates an example of such mapping. The composites are part of the TuscanySCA-Tours scenario. It deals with travel booking, and it depends on the Payment composite to handle payments made by the customer. The CreditCardPayment composite is a service provider that processes credit card transactions.

In this example, three Tuscany nodes are used to run the applications. Two nodes are run on one JVM (JVM1). Node1 hosts the components from the SCATours composite whereas Node2 hosts components from the Payment composite. Node3 is started on a different JVM (JVM2) to run components from the CreditCardPayment composite.

Tuscany is quite flexible in how it maps the domain composite to different node topologies. For example, you can run all three composites on one single node or run

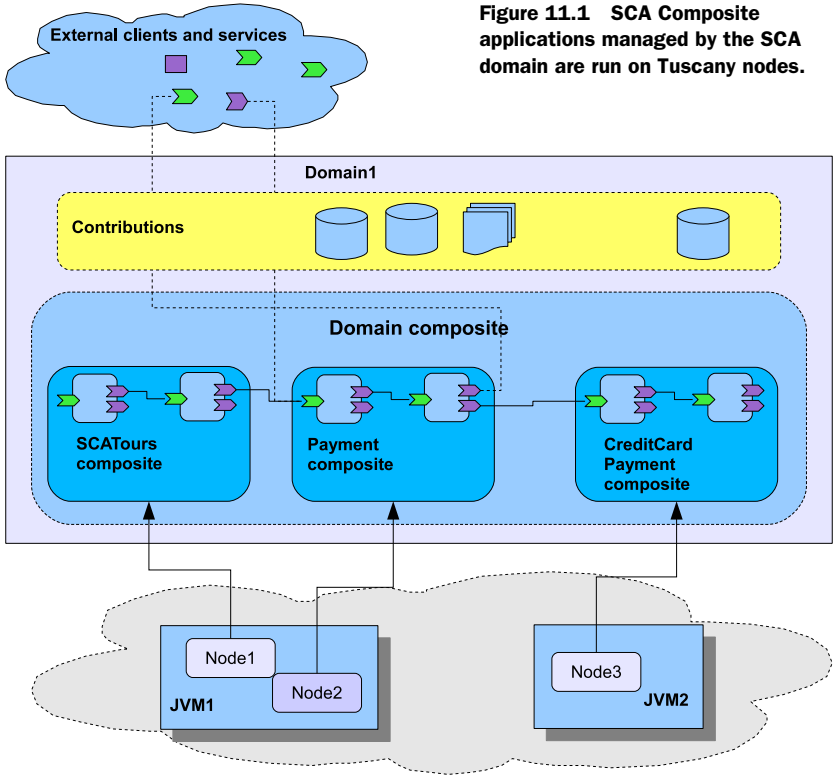


Figure 11.1 SCA Composite applications managed by the SCA domain are run on Tuscany nodes.

each of the composites on a different node. These mappings can be described in the Tuscany node configuration.

11.1.2 Tuscany node configuration

A Tuscany node is configured to run a composite application, which could be the entire SCA domain composite or a subset of this composite. The configuration captures the following information:

- A collection of contributions that are required to run the components on the node
- One or more deployable composites that describe the components
- A list of binding base URIs that tells Tuscany what addresses are used to publish the service endpoints

To map an SCA domain to Tuscany nodes, you can partition the domain composite into one or more groups of components (denoted by composites) that are suitable to be run together. Each group of composites is configured as a Tuscany node.

SCA components typically depend on artifacts from SCA contributions such as the interface, Java implementation class, or XML schema. A list of installed contributions will be selected from the SCA domain for a node.

You can also specify the default configuration for the communication infrastructure that a node provides. For example, a service with `binding.ws` can be published on one node as http://node1:8080/component1/service1/binding1 and on another node as http://node2:8085/component1/service1/binding1. The node configuration defines which HTTP ports to use by default.

The node configuration can be manually defined, automatically discovered, or provided by the domain manager online or offline, depending on how the node is run. We'll explore the various options in the next sections.

11.1.3 Hosting options for a Tuscany node

The hosting environment for Tuscany controls how to configure, create, start, and stop a node. Each environment uses various approaches to handle the following aspects:

- How to create a node configuration
  - From a configuration file
  - From the domain manager
  - From a list of command-line arguments
  - From what's in the environment (classpath, repository, and so on)
- How to control the lifecycle of a node
  - Start/stop the node using a launcher
  - Start/stop the node explicitly within the embedding application
  - Start/stop the node from a lifecycle listener of the host environment, such as the `ServletFilter` or `ServletContextListener` or the OSGi Bundle Activator, Service Listener, or Bundle Listener
- Are there SCA wires that go across components that are hosted on different nodes?
  - Components in the node access services only in the same node.
  - Components that are wired by SCA run on different nodes.
- How to find endpoint descriptions of other SCA services in the same domain but running on a different node
  - From a domain registry
  - From a local file
- How to initiate component interactions
  - Access the SCA services from a remote client
  - Access the SCA services locally
  - From a component that's early initialized
  - From a launcher that knows the client component

These options are illustrated in figure 11.2.

As shown in figure 11.2, the SCA assembly represented by the domain composite (A) can be partitioned into groups of components. This is typically done by using one

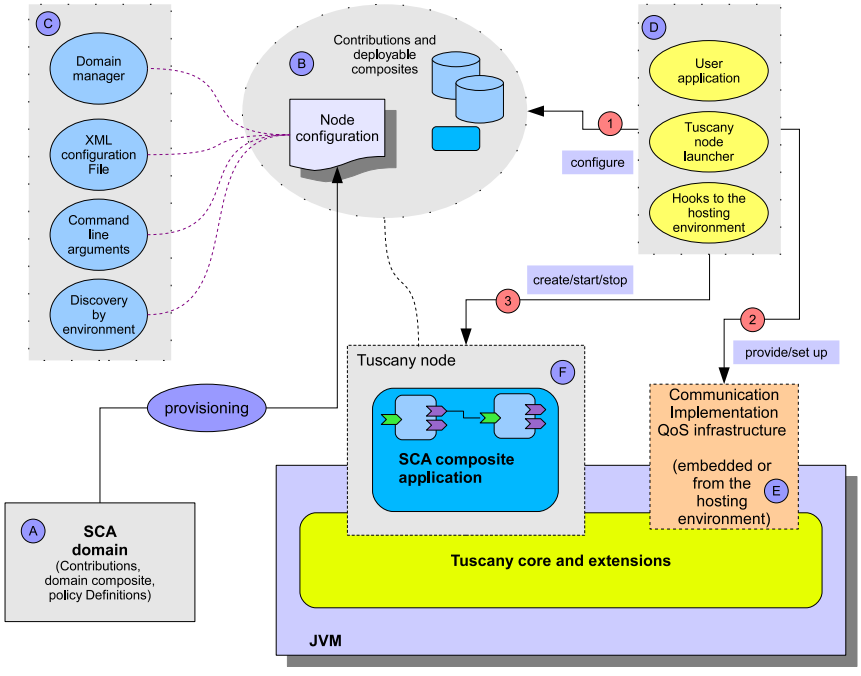


Figure 11.2 The SCA domain is mapped to a set of Tuscany nodes to be run in different environments, which can use different ways to configure, create, start, and stop a node.

or more composites that define the components. You can also use a deployment composite to regroup the components. Each group will be hosted by a Tuscany node (F). You'll define the domain/node mapping using the node configurations (B). You can bootstrap the Tuscany runtime in different environments (D) to provide the infrastructure (E) for one or more nodes. Depending on the hosting environment, you can provide the configuration for a node using different mechanisms (C) and then create and start nodes to run the components.

Tuscany provides the `SCANodeFactory` and `SCANode` APIs so that different hosting environments can use them to create an `SCANode` from a node configuration and start or stop the `SCANode` to run the composite application. Each form of hosting option for Tuscany ultimately calls the `SCANodeFactory` and `SCANode` APIs.

In the following sections we'll look at the hosting options Tuscany provides to help you understand and decide which one best fits your needs.

11.2 Running Tuscany standalone

Let's start with the simplest form: running the SCA application from the command line. Tuscany comes with a launcher that supports many implementation and binding types out of the box in a standalone JVM process. You can start your SCA application

using a Java command, passing the list of SCA contributions as a parameter. No external application servers or installation steps are required. Let's give it a try with the creditcard-payment-jaxb contribution. To run it with the Tuscany node launcher from the command line, type

```
java -jar <TUSCANY_HOME>\modules\tuscany-node-launcher-<version>.jar
     creditcard.composite scatours-contribution-creditcard-payment-jaxb.jar
```

The precise location of the contribution JAR file will vary depending on whether you have the src or binary sample distribution. When this command is issued, the Tuscany node launcher creates a node configuration using the arguments that are passed in for the deployable composite and contributions. The launcher discovers the JARs in the Tuscany distribution and uses them to bootstrap the Tuscany runtime. The configured node is created and started in order to run the application. You can verify it by pointing your browser to http://localhost:8082/CreditCardPayment?wsdl to get the WSDL document for the service.

The standalone Tuscany runtime can embed other frameworks such as Tomcat/Jetty, Axis2, and ActiveMQ to provide protocol stacks such as HTTP, SOAP, or JMS for SCA bindings. It can also embed Spring for implementation.spring and Apache ODE for implementation.bpel. The node launcher is illustrated in figure 11.3.

The launcher operates as a container where SCA components can expose services with certain protocols using bindings. In the credit card payment application, a web service is published on behalf of the `binding.ws` configuration for the CreditCardPayment service. There's no need to start an HTTP server manually.

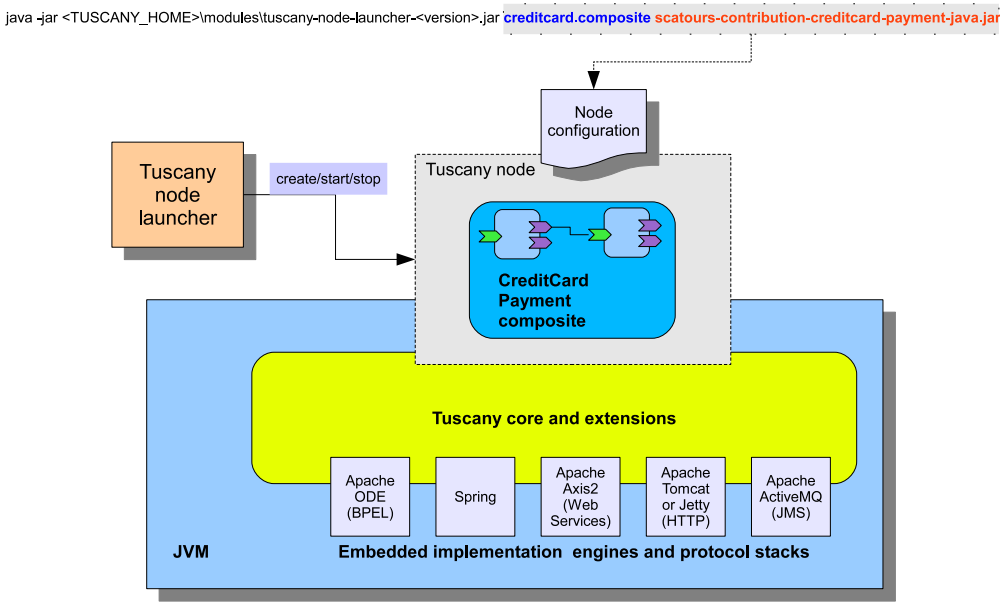


Figure 11.3 The Tuscany node launcher bootstraps the Tuscany runtime and creates a node to load and run the composite application.

It's also possible to run a composite application with multiple contributions. For example, you can run the Payment and CreditCardPayment composites as follows:

```
java -jar <TUSCANY_HOME>\modules\tuscany-node-launcher-<version>.jar -
     scatours-contribution-payment-java.jar scatours-contribution-creditcard-
     payment-jaxb.jar
```

In this case you omit the deployable composite (using – as the first argument) and use two JARs as the contributions.

The Tuscany node launcher provides a simple way to run SCA composite applications directly from one or more contributions in a standalone JVM process. It provides the infrastructure out of the box for the common implementation and binding types without extra installation and configuration of the stacks. This hosting option is useful if the application is built using SCA to expose services with protocols that can handle requests from external clients. Because the launcher is running as a standalone JVM, you'll need to invoke SCA components from a remote client.

The Tuscany node launcher shields user applications from setting up the Tuscany runtime and calling Tuscany APIs to control the node. There are also use cases where programmatic access to the Tuscany runtime is required.

11.3 Running Tuscany using APIs

Certain types of application need to invoke Tuscany programmatically by using Tuscany as a library for embedding an SCA runtime. Here are some examples:

- The application is managed or launched by another means such as the OSGi framework, Eclipse RCP, or JUnit runner.
- The application needs to access SCA services locally so that SCA components can be triggered without a remote client.
- The application needs to start or stop Tuscany programmatically for testing and debugging purposes.

Tuscany provides a simple node API to bootstrap the runtime and manage nodes. It gives users, developers, and embedders the ability to interact with Tuscany with full control. Figure 11.4 shows a simple customized launcher from the TuscanySCATours application to demonstrate how to interact with Tuscany node APIs.

As illustrated in figure 11.4, the typical sequence of Tuscany API calls is as follows:

1. Create one or more `SCAContribution` instances using the contribution URI and location URL.
2. Create an instance of `SCANodeFactory` and use it to create an instance of `SCANode` from the deployable composite and one or more `SCAContribution` instances.
3. Start the `SCANode`.
4. Get a local proxy for the given SCA component.
5. Invoke the service to perform business logic.
6. Stop the `SCANode`.

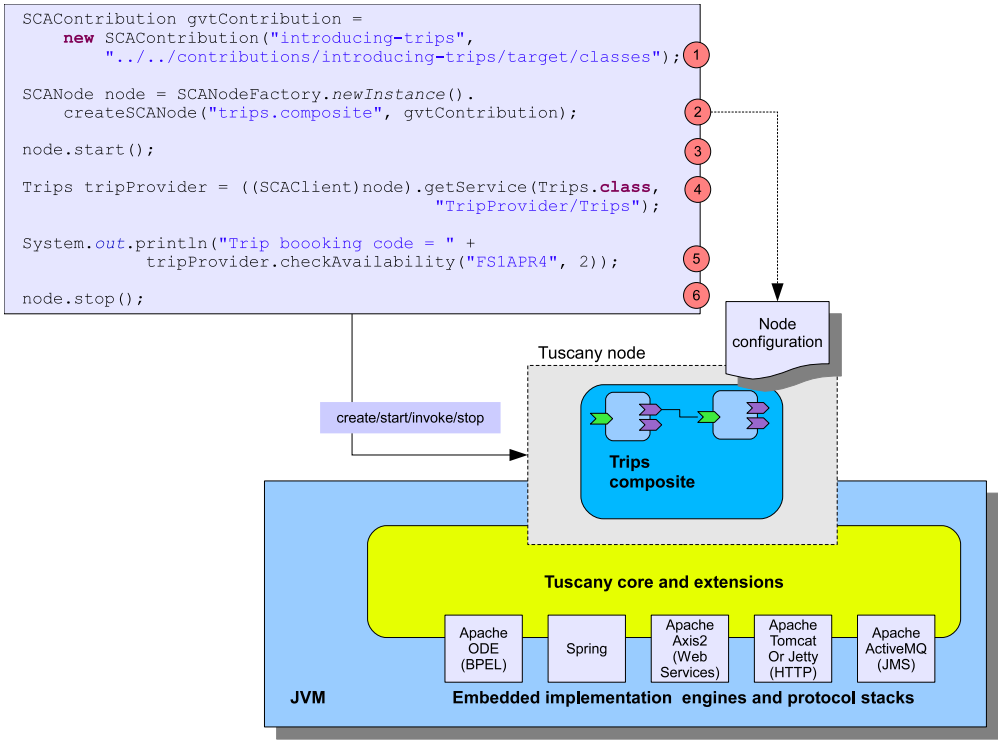


Figure 11.4 Tuscany nodes can be programmatically configured and created. Application code uses the node APIs to start the node, look up a service, invoke it, and stop the node.

Steps 1 and 2 can be triggered by hooking the methods to lifecycle events of the hosting environment. The following example uses JUnit 4 annotations to start Tuscany with a local contribution under the target/classes folder, test the service calls, and then stop.

Listing 11.1 A JUnit test case that calls Tuscany APIs

```
public class CreditCardPaymentTestCase {
    private static SCANode node;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        node = SCANodeFactory.newInstance().createSCANode(null,
            new SCAContribution("creditcard", "./target/classes"));
        node.start();
    }

    @Test
    public void testCreditCardPayment() {
        SCAClient client = (SCAClient)node;
        CreditCardPayment cc = client.getService(
            CreditCardPayment.class, "CreditCardPayment");
```

1 **Start up Tuscany node**

2 **Look up CreditCardPayment SCA service**

```
        ObjectFactory objectFactory = new ObjectFactory();
        CreditCardDetailsType ccDetails =
            objectFactory.createCreditCardDetailsType();
...

        try {
            System.out.println(cc.authorize(ccDetails, 100.00f));
        } catch (AuthorizeFault_Exception e) {
            System.err.println("Fault: " +
                               e.getFaultInfo().getErrorCode());
        }
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        if (node != null) {
            node.stop();
            node = null;
        }
    }
}
```

3 Use SCA service

4 Shut down Tuscany node

The JUnit code uses a method annotated with `@BeforeClass` to start the Tuscany node 1. One test method looks up the `CreditCardPayment` SCA service 2 and invokes the `authorize()` method 3. Finally, the Tuscany node is shut down by the method annotated with the `@AfterClass` annotation 4.

As an alternative to using a standalone application, many hosting environments provide a means to manage applications, including deployment and lifecycle. It's often desirable to leverage these capabilities to make Tuscany's SCA execution environment available for such applications. Java EE web containers are one of the popular technologies for hosting applications. Let's explore how to integrate Tuscany with web applications.

11.4 Running Tuscany with web applications

Web applications have evolved beyond the HTML-based presentation layer. More and more web applications integrate with backend enterprise services to provide business services over HTTP. With the emergence of cloud computing and the availability of cloud infrastructure such as Amazon EC2 and the Google App Engine platform, it's often desirable to enable service construction and composition in a simple and flexible way so that you can mash up business applications.

What help can Tuscany and SCA provide for web applications? You can develop composite applications using the SCA programming model to assemble components, access existing web services or EJB session beans, and expose SCA component services over HTTP-based protocols such as Web Services, JSON-RPC, and Atom or RSS feeds. SCA can leverage the security and transaction infrastructure to enforce quality of service (QoS) in a consistent fashion. In this section we'll look at how to package Tuscany SCA applications inside Java EE web application archive (WAR) files.

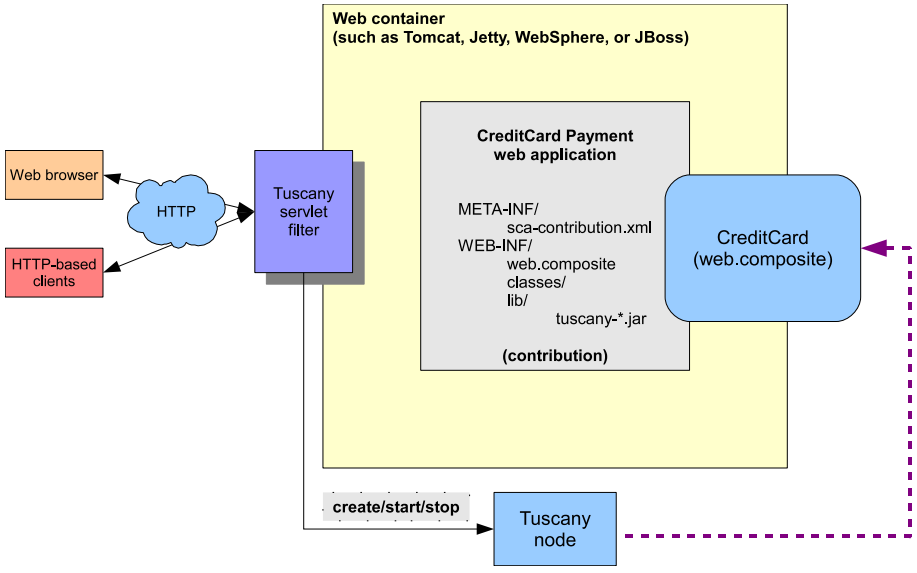


Figure 11.5 Tuscany runtime JARs are packaged in the WAR to allow the web application to run as an SCA contribution.

To enable SCA within web applications, Tuscany provides a simple mechanism for packaging the runtime JARs together with the application artifacts into WAR files. An SCA composite is packaged inside a WAR file alongside the application artifacts and the Tuscany JARs and their dependencies. You can directly deploy the WAR file to standards-based web containers such as Tomcat, Jetty, and IBM WebSphere. Container-specific deployment descriptor files may also be added.

As we mentioned in chapter 8, a Tuscany-provided servlet filter is configured to start up or shut down the Tuscany runtime when the web application is started or stopped. The Tuscany filter is also responsible for dispatching HTTP requests to HTTP-based bindings such as Web Services, JSON-RPC, and Atom/RSS Feed bindings. Figure 11.5 shows how the Tuscany servlet filter bridges between the web application framework and SCA.

Let's look at the structure of the Tuscany-enabled web application. There are a few important files and folders in the WAR, as shown in table 11.1.

Table 11.1 Important files in a Tuscany SCA–enabled web application

File in the WAR	Description
META-INF/sca-contribution.xml	Defines the deployable composites for a given contribution
WEB-INF/web.composite	Describes the assembly for the web application
WEB-INF/web.xml	Configures the web application with a Tuscany servlet filter
WEB-INF/lib	Contains all the required Tuscany JARs and their dependencies

As shown next, you configure the web.xml file with a Tuscany servlet filter, which is responsible for managing the node and dispatching HTTP requests to the corresponding SCA service-binding listeners.

11.4.1 Configuring WEB-INF/web.xml

To enable a web application with Tuscany SCA, you'll need to configure a servlet filter from Tuscany in the WEB-INF/web.xml file. The following listing shows an example.

Listing 11.2 Sample web.xml for a Tuscany-enabled web application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3/
     /EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

  <display-name>Apache Tuscany Calculator Web Service Sample</display-name>
  <filter>

     <filter-name>tuscany</filter-name>
     <filter-class>
      org.apache.tuscany.sca.host.webapp.TuscanyServletFilter
     </filter-class>
  </filter>

  <filter-mapping>
     <filter-name>tuscany</filter-name>
     <url-pattern>/*</url-pattern>
  </filter-mapping>
</webapp>
```

Configure Tuscany servlet filter

Filter intercepts all HTTP requests

The filter will be triggered upon the first HTTP request to this web application and will create and start a Tuscany node. The node instance is set in the servlet context as an attribute so that it can be accessed later by JSPs and servlets. When the web application is stopped, the filter is called to stop and destroy the Tuscany node.

11.4.2 Customizing class loading policy

The Tuscany runtime JARs and their dependencies are packaged in the WEB-INF/lib folder within the WAR file. Some third-party JARs such as the stax-api and Woodstox implementations might conflict with those from the web container. The good news is that most Java EE application servers, such as Apache Geronimo and IBM WebSphere, allow different settings for the class-loading policy. To ensure that the Tuscany runtime and its dependency classes will be loaded from the WEB-INF/lib folder, you'll need to configure the class-loading policy so that it creates an isolated environment for the Tuscany runtime without interference from other JARs shipped by the application server (for example, the Axis2 JARs shipped by Geronimo). Listing 11.3 gives an example of the Geronimo deployment descriptor. The `<d:inverse-class-loading>` element is used to indicate that classes from the web application should be loaded in preference to classes provided by the application server.

Listing 11.3 Sample WEB-INF/geronimo-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://geronimo.apache.org/xml/ns/j2ee/web-2.0"
    xmlns:d="http://geronimo.apache.org/xml/ns/deployment-1.2">
    <d:environment>
        <d:moduleId>
            <d:groupId>org.apache.tuscany.sca</d:groupId>
            <d:artifactId>sample-calculator-ws-webapp</d:artifactId>
            <d:version>1.0-SNAPSHOT</d:version>
            <d:type>war</d:type>
        </d:moduleId>
        <d:inverse-classloading />
    </d:environment>
</web-app>
```

Please note that Tomcat, by default, has the child-first class-loading policy (where the classes packaged in the WAR file take precedence) for web applications, so no extra class loader configuration is needed.

11.4.3 Deploying Tuscany-enabled web applications

After the WAR file is created, you can deploy it to the web container. For Tomcat, you can copy it into the webapps folder. When the web container starts, the Tuscany node will be created and started for the web application when the application is started.

Using this approach, the Tuscany runtime is enclosed within the web application and isolated from other web applications. You can deploy more than one Tuscany-enabled web application in the same web container without them interfering with each other.

Hosting Tuscany using a web application provides a simple but powerful option to run SCA applications within a web container. It's easy to produce a WAR file. With SCA, web applications can compose business logic, invoke external services, and expose services to a Web 2.0 frontend or to business partners. Meanwhile, you'll get the HTTP-based binding and policy infrastructure for free because they're part of the web container. Tuscany-enabled applications can leverage the Web Services, JSON-RPC, and Atom bindings out of the box.

So far we've explored the hosting options for Tuscany nodes that have been pre-configured and resolved. These nodes don't rely on the information from other nodes in the same SCA domain. In the following section, you'll learn how to configure and run distributed nodes.

11.5 Configuring distributed nodes

In chapter 3, we created an SCA domain from scratch using the Tuscany Domain Manager GUI and ran a distributed Tuscany application using the Tuscany domain manager. The Domain Manager GUI saves the domain configuration as a number of XML files. As an alternative to using the Domain Manager GUI to create and update the configuration files, you can edit the domain configuration files directly. Editing the configuration files can be a simpler way of creating or modifying a domain configuration

than using the Domain Manager GUI, especially if you need programmatic control of the contents of these files. In addition, editing these files provides additional flexibility (for example, when configuring bindings) that isn't possible when using the Domain Manager GUI as it stands at the time of writing.

To help you understand how to edit these configuration files, this section describes what the files contain. After creating or editing the files as described in the rest of this section, you can run the Tuscany domain in exactly the same way as we described in chapter 3 except for omitting the steps that refer to using the Domain Manager GUI.

There are two important words of warning before we dive into the details of the configuration files: *Be careful!* The format and contents of the configuration files are straightforward, and it isn't difficult to create or modify them using a text editor. But if you make a mistake when doing this (even the tiniest typo), the domain manager won't work properly. It's likely to malfunction in mysterious ways with no indication of what's causing the problem. So if things aren't working correctly after you've edited the configuration files, go back and double-check the contents of these files. If you can't see a problem, check them again!

In the following sections, we'll walk you through the manual steps required to define an SCA domain, including installed contributions, deployed composites, runtime nodes, and base URIs of published services for each binding type. The files are needed in Tuscany to run a distributed composite application within an SCA domain. We'll start by defining which contributions are part of the domain.

11.5.1 Defining the contents of the domain code repository

The domain code repository is defined by the contents of the workspace.xml file. The following listing shows an example.

Listing 11.4 Example workspace.xml file for a domain with three installed contributions

```
<?xml version="1.0" encoding="UTF-8"?>
<workspace xmlns="http://tuscany.apache.org/xmlns/sca/1.0"
        xmlns:ns1="http://tuscany.apache.org/xmlns/sca/1.0">
    <contribution location="file:../contributions/
    ➥ introducing-tours/target/scatours-introducing-
    ➥ tours.jar" uri="introducing-tours"/>
    <contribution location="file:../contributions/
    ➥ introducing-trips/target/scatours-introducing-
    ➥ trips.jar" uri="introducing-trips"/>
    <contribution location="file:../contributions/
    ➥ introducing-client/target/scatours-introducing-
    ➥ client.jar" uri="introducing-client"/>

    <contribution location="file:./cloud"
        uri="http://tuscany.apache.org/cloud"/>
</workspace>
```

(1) Contributions installed in domain

(2) Special entry for internal use

In listing 11.4, each contribution installed in the domain code repository is defined by a `<contribution>` element (1) within the `<workspace>` element. In this example the

contribution locations are specified as `"file:"` URLs relative to the domain manager's execution directory. There's also a special `<contribution>` element ❷ that's used internally by the domain manager.

11.5.2 Specifying the deployed composites

The deployed composites are listed in the domain.composite file. An example of this file is shown here.

Listing 11.5 Deployed composites are listed in the domain.composite file.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite local="true" name="domain"
           targetNamespace="http://tuscany.apache.org/xmlns/sca/1.0"
           xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:ns1="http://www.osoa.org/xmlns/sca/1.0">
    <include name="ns2:Trips" uri="introducing-trips"
           xmlns:ns2="http://goodvaluetrips.com/"/>
    <include name="ns2:Tours" uri="introducing-tours"
           xmlns:ns2="http://tuscanyscatours.com/"/>
    <include name="ns2:Client" uri="introducing-client"
           xmlns:ns2="http://client.scatours/"/>
</composite>
```

Listing 11.5 is the domain manager's definition of the SCA domain composite. It contains `<include>` elements for all the composites that have been deployed to the domain. The `uri` attribute on the `<include>` element is a Tuscany extension to SCA and is used to specify the URI of the contribution that contains the composite.

11.5.3 Defining the nodes in the execution cloud

The last file in the domain manager's working directory is called cloud.composite, and it contains a list of execution nodes for deployed composites. The following listing shows an example of this file.

Listing 11.6 Execution nodes for deployed composites

```
<?xml version="1.0" encoding="UTF-8"?>
<composite local="true" name="cloud"
           targetNamespace="http://tuscany.apache.org/xmlns/sca/1.0"
           xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:ns1="http://www.osoa.org/xmlns/sca/1.0">
    <include name="ns2:TripsNode"
           uri="http://tuscany.apache.org/cloud"
           xmlns:ns2="http://tuscany.apache.org/cloud"/>
    <include name="ns2:ToursNode"
           uri="http://tuscany.apache.org/cloud"
           xmlns:ns2="http://tuscany.apache.org/cloud"/>
    <include name="ns2:ClientNode"
           uri="http://tuscany.apache.org/cloud"
           xmlns:ns2="http://tuscany.apache.org/cloud"/>
</composite>
```

Listing 11.6 is the domain manager's definition of the execution cloud (distributed network of execution nodes) for the domain. Each execution node is represented by an `<include>` element giving the name of the node. The `uri` attribute and XML namespace of an execution node are always `"http://tuscany.apache.org/cloud"`.

11.5.4 Configuring bindings for the nodes in the execution cloud

In addition to the three configuration files in the working directory of the domain manager, the working directory has a cloud subdirectory, which contains a file for each execution node. These files have a file extension of .composite and a filename corresponding to the node name. The next listing shows the TripsNode.composite file, which represents the `TripsNode` execution node.

Listing 11.7 The TripsNode.composite file representing the `TripsNode` execution node

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:t="http://tuscany.apache.org/xmlns/sca/1.0"
           xmlns:c="http://tuscanyscatours.com/"
           targetNamespace="http://tuscany.apache.org/cloud"
           name="TripsNode">
    <component name="TripsNode">
        <t:implementation.node
            uri="introducing-trips"
            composite="c:Trips" />
        <service name="Node">
            <binding.sca uri="http://localhost:8082" />
            <binding.ws uri="http://localhost:8082" />
            <binding.http uri="http://localhost:8082" />
            <binding.jsonrpc uri="http://localhost:8082" />
            <binding.atom uri="http://localhost:8082" />
        </service>
    </component>
</composite>
```

1 **Composite that this node will run**

2 **Runtime endpoint information**

In listing 11.7 the TripsNode composite contains a single special component that's also called TripsNode to describe the node configuration using XML. This component has an implementation type of `implementation.node`.[1] The `<implementation.node>` element has a `composite` attribute specifying the name of the composite 1 that will be run by this execution node and a `uri` attribute identifying the contribution that contains this composite. There's a single service named Node with a `binding` element 2 for each binding used by services running on this node. This `binding` element has a `uri` attribute with the protocol, host, and port used by the node's runtime to expose services with this binding type.

[1] The `<implementation.node>` element is defined in a Tuscany-specific XML namespace instead of the regular SCA namespace because it's a Tuscany extension that isn't part of the SCA specifications.

The Domain Manager GUI always creates node configuration files with binding elements for `binding.sca`, `binding.ws`, `binding.http`, `binding.jsonrpc`, and `binding.atom`, with all of these having the same protocol, host, and port information as shown in listing 11.7. However, the domain manager runtime can handle any combination of `binding` elements in the node configuration file, with either the same settings for protocol, host, and port for all bindings or different settings for protocol, host, and port for different bindings. Any bindings that aren't used by this execution node don't need to appear in its Node service configuration. For example, the Node service could be defined as follows:

```
<service name="Node">
    <binding.sca uri="http://node1.mydomain.com:8091" />
    <binding.ws uri="http://node2.mydomain.com:8092" />
    <binding.foo uri="foo://node3.mydomain.com:8093" />
</service>
```

This means that any service endpoints using `binding.sca` are hosted on port 8091 of `node1.mydomain.com`, any service endpoints using `binding.ws` are hosted on port 8092 of `node2.mydomain.com`, and any service endpoints using `binding.foo` (a user-defined binding) use the `foo:` protocol and are hosted on port 8093 of `node3.mydomain.com`.

The domain manager uses the node configuration endpoint information to customize the service and reference bindings in the deployed composites that are passed to the execution nodes. In this way all of the services of the nodes in the distributed runtime will expose unique endpoints and can communicate with each other. You saw some examples of doing this in section 3.2.5, so we won't repeat them here.

In this section you've seen how you can modify or create a domain configuration without going through the Domain Manager GUI. This can be useful either for automating the domain configuration process or for providing more flexibility than the Domain Manager GUI can support.

Next, we'll look at how the Tuscany runtime can be embedded into other containers such as a Java EE container.

11.6 Embedding Tuscany with a managed container

We've explored several options provided by Tuscany out of the box for running SCA composite applications. With this understanding of the responsibilities for a host environment and how it interacts with Tuscany, it's time to add a bit more information for those who want to embed Tuscany in their own managed environment, such as an OSGi framework or Java EE container. Let's recall the roles of a host environment:

1. *Configure the node*—Find out the deployable composites and required contributions that constitute the group of components to be run on this node.
2. *Provide the infrastructure for a node*—Plug in host-* modules to provide hooks for binding protocols, QoS system services, and implementation engines/frameworks.

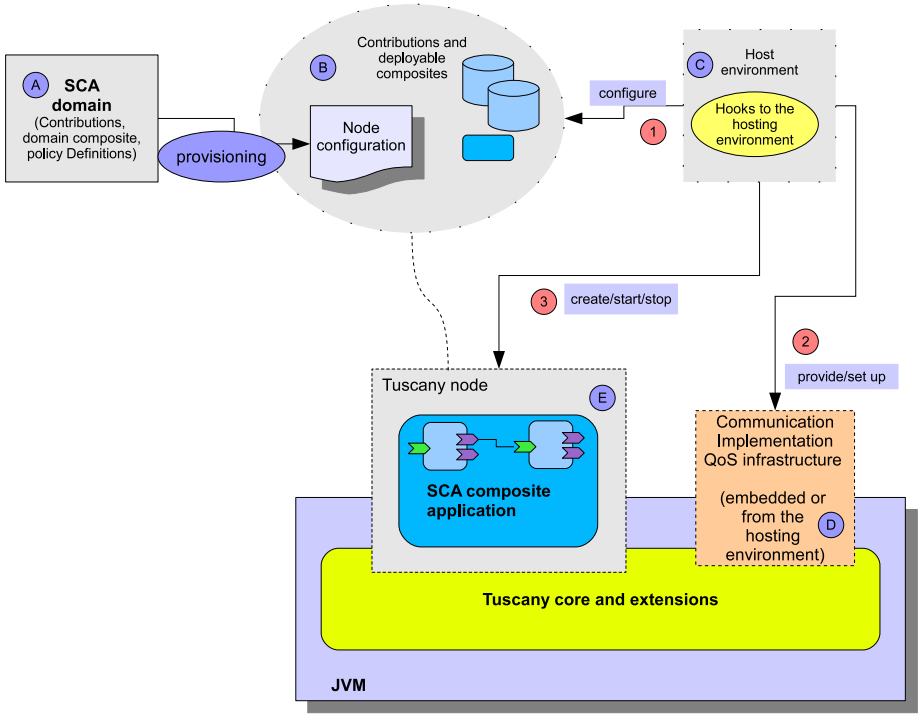


Figure 11.6 A managed container that embeds Tuscany and provides infrastructure for SCA. It calls Tuscany APIs to manage the configuration, creation, and lifecycle of the node.

3 *Control the lifecycle of a node*—Leverage the `SCANodeFactory` and `SCANode` APIs to create, start, or stop a node.

These responsibilities are illustrated in figure 11.6.

To embed Tuscany, you'll typically need to plug code into the hosting environment. First, you'll choose how to group components from the SCA domain composite to configure a node. This configuration information can be retrieved from the domain manager or can be loaded from a local file. Second, a managed environment typically has a way to control the lifecycle of the applications it manages. For example, an OSGi bundle listener or web application context listener are able to listen for the addition or removal of applications. The host can associate Tuscany nodes with such a boundary and call node APIs to create, start, or stop the node appropriately.

Tuscany binding, implementation, and policy features can be provided in different ways by different hosting environments. You can build new extensions for the Tuscany extension APIs to provide these features in host-specific ways if required. For example, a web container host can provide an HTTP service as a servlet, whereas an OSGi container can use the OSGi HTTP service. Chapter 13 discusses the extension APIs in more detail.

11.7 Summary

As you've learned in this chapter, SCA applications can be run in different environments using the various hosting options that Tuscany provides. The environment can be as simple as a command-line launcher or a web application where the SCA application will run in a single node in a single JVM.

For more complex and distributed scenarios, the Tuscany domain manager provides a mechanism for deploying SCA applications in a distributed environment within an SCA domain. It achieves this by configuring the composite files that are deployed to a set of distributed nodes.

If none of the hosting options provided by Tuscany suit your needs, you can also embed Tuscany in your own container to add the power of SCA into your application server.

Now that you understand how to start the Tuscany runtime in various environments, let's move on and pull some of the strands of the book together by taking a look at the TuscanySCATours travel-booking application in all its glory.

12 A complete SCA application

This chapter covers

- Running the TuscanySCATours application in a single Tuscany node
- Building the application from separate SCA composites
- Running the application in a distributed SCA domain
- Hints and tips for building composite applications

We've used the TuscanySCATours travel-booking application throughout the book to demonstrate the various features of the Tuscany SCA Java runtime. In this chapter we'll pull all the various parts together and run the complete application. The TuscanySCATours travel-booking application may not be much like your application, or even a realistic application as it stands, but showing all of the parts running together allows us to stand back from the details and see an SCA application doing something more than printing test output on the console.

On occasion we've presented different versions of the same component. For example, chapters 5 and 6 use different versions of the Payment component to demonstrate Java language, BPEL, Spring, and script implementation types. Here we'll select one version of each component to build the full application. We'll look at how the components in the application are organized and review some of the interesting features that we've included as the book has developed.

This chapter is also an opportunity for us to share some hints and tips about putting SCA applications together. These mostly focus on the nonfunctional issues of building SCA applications, such as how best to organize and test contributions.

To start off we'll deploy and run the complete application in a single Tuscany node using the simple fullapp launcher. We'll then look at how we brought the application together and describe each of the composites in the application. This will take up most of the space in this chapter because we'll talk about the eight different composites that the application uses. After this we'll run the same application in a distributed domain using multiple nodes. This uses the fullapp-domain and fullapp-nodes launchers. We'll finish off by discussing some of the lessons we learned when building the application. Let's get set up and get the application running.

12.1 Getting ready to run the application

The TuscanySCATours travel-booking application is a sample application that allows you to search for and book trips. The application features access by the user via a single web page that provides search input, search results, a shopping cart, and a checkout button. It's not a real application in that many important online retail features are missing—a full security solution, for example. The application does, however, provide an example of how SCA can be used to partition and assemble a reasonably complex application.

The sample code comes with a single JVM launcher for the full TuscanySCATours application. In this chapter we'll run the sample on Windows (although it will run on other platforms that the Java language supports). You can find it in the following file: /launchers/fullapp/src/main/java/scatours/FullAppLauncher.java. This file has everything in it to load all of the contributions that make up the full application. The following listing shows the launcher code..

Listing 12.1 Loading all of the contributions that form the full travel application

```
public class FullAppLauncher {

    public static void main(String[] args) throws Exception {
        SCANode node =
            SCANodeFactory.newInstance().createSCANode(null,
                      locate("common"),
                      locate("currency"),
                      locate("hotel"),
                      locate("flight"),
                      locate("car"),
                      locate("trip"),
                      locate("tripbooking"),
```

```
                        locate("travelcatalog"),
                        locate("payment-spring-policy"),
                        locate("creditcard-payment-jaxb-policy"),
                        locate("shoppingcart"),
                        locate("scatours"),
                        locate("fullapp-ui"),
                        locate("fullapp-coordination"),
                        locate("fullapp-currency"),
                        locate("fullapp-packagedtrip"),
                        locate("fullapp-bespoketrip"),
                        locate("fullapp-shoppingcart"));
            node.start();

            System.out.println(
                "Point your browser at - http://localhost:8080/scatours/ ");
            System.out.println("Node started - Press enter to shutdown.");
            try {
                System.in.read();
            } catch (IOException e) {
            }
            node.stop();
        }
}
```

In listing 12.1 you can see that the `locate()` utility operation is used to find each of the required contributions instead of providing the full path to the contribution. The `locate()` utility isn't part of Tuscany. It was written for this book sample to make the launcher code more compact.

The contributions shown here are a subset of the contributions you'll see in the sample's contributions directory. As we said in the introduction to this chapter, there are contributions that demonstrate particular SCA features that are used by other chapters but aren't used in the full application. Also note that not all of the contributions loaded by this launcher contain a composite file. Some of the contributions just allow us to share artifacts such as Java class files or WSDL files between various parts of the application. For example, the common contribution contains the basic Java interfaces shared among other contributions.

Running the launcher shown in figure 12.1 is straightforward once your environment is set up to run Tuscany. Appendix A provides the necessary details. The README file provided with the travel-booking sample provides instructions on how to run each launcher. For example, if you have the source code distribution of the travel-booking sample, you can run the fullapp launcher using the Ant build.xml file provided in the /launchers/fullapp directory. To run the build.xml file enter the following:

```
cd launchers/fullapp
ant run
```

Once the travel-booking application is running, it prints messages to the console that indicate contributions are being loaded. The launcher prints the following message when initialization is complete:

```
Point your browser at - http://localhost:8080/scatours/
Node started - Press enter to shutdown.
```

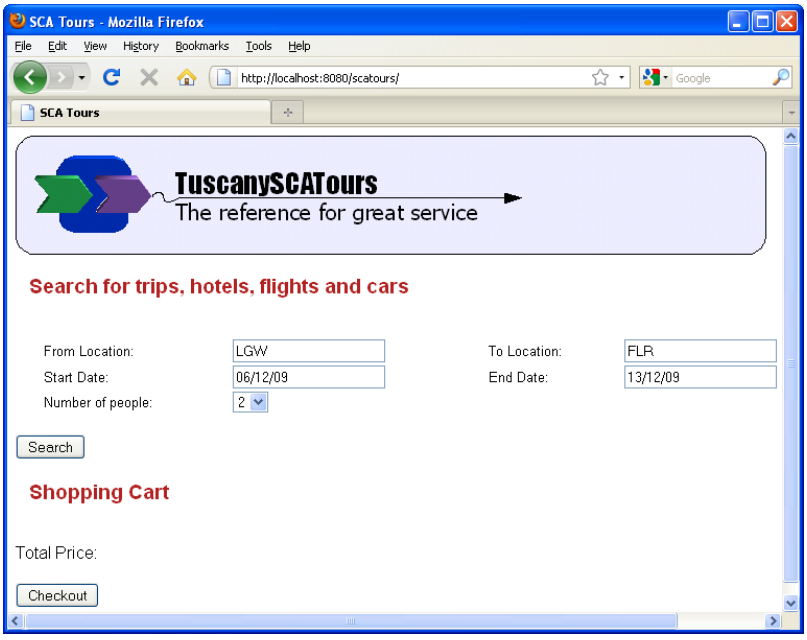


Figure 12.1 The front page of the TuscanySCATours travel-booking application

Try the application now by pointing a web browser at http://localhost:8080/scatours/. The page you'll see is shown in figure 12.1.

You can search for trip packages and trip components using the Search button. The trips showing in the search results list can be selected and added to the shopping cart. Finally, you can choose to buy the trips and check out. This will clear the shopping cart.

When you've finished, press a key in the window where the launcher was started, and the application will stop running. Let's review all of the contributions you saw being loaded by the launcher.

12.2 Assembling the travel-booking application

In various chapters of the book we've shown different versions of the same components. Here we'll choose one version of each component in the application and wire them all together. Figure 12.2 shows the first part of the application and includes the name of the technology used to implement each component.

In figure 12.2 the numbers in brackets indicate which HTTP port each composite occupies when using HTTP-based bindings such as the Web Services or JSON-RPC binding. Note also that the names of the composites in the diagram match the names of the contributions that can be found in the sample contributions directory.

Figure 12.2 shows how the travel-booking application is composed of smaller, functional components using SCA's flexible configuration and wiring.

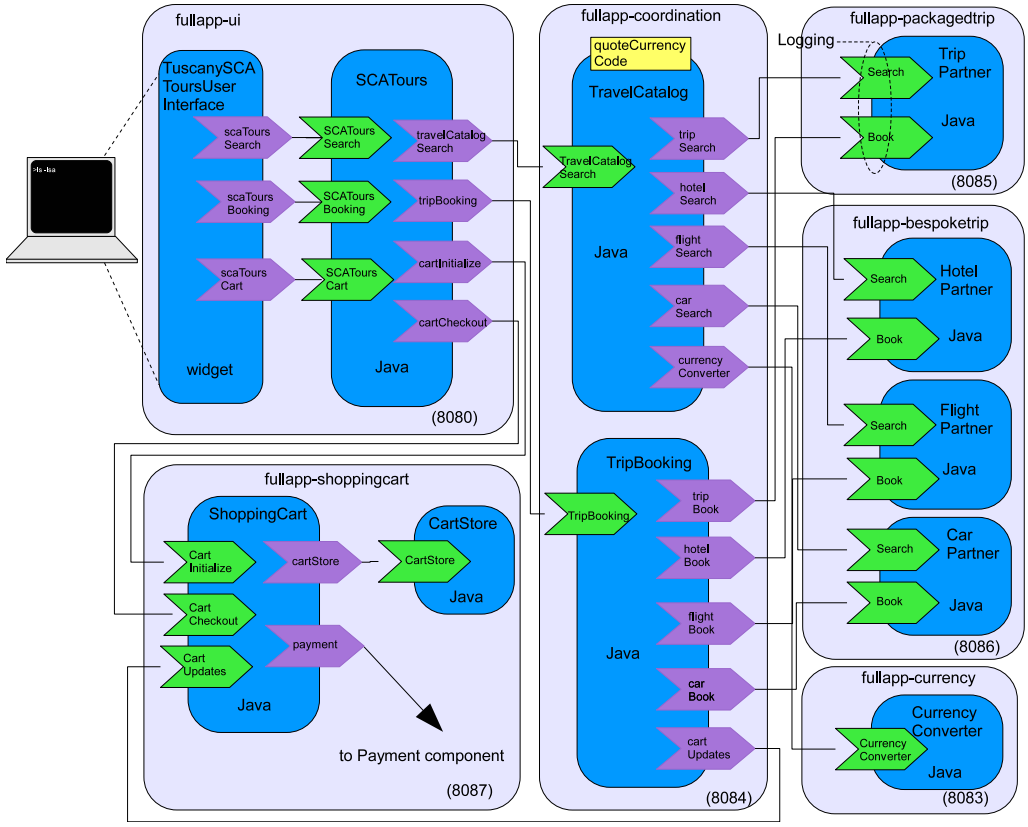


Figure 12.2 The first part of the TuscanySCATours travel-booking application up to and including the ShoppingCart component

SCA components are collected into composites such as `fullapp-shoppingcart` and `fullapp-ui`. Each composite is a unit of deployment and can run on a separate Tuscany node. We've split the application diagram into two for readability. The second part of the application includes the payment components from the payment-spring-policy and creditcard-payment-jaxb-policy sample contributions, as shown in figure 12.3.

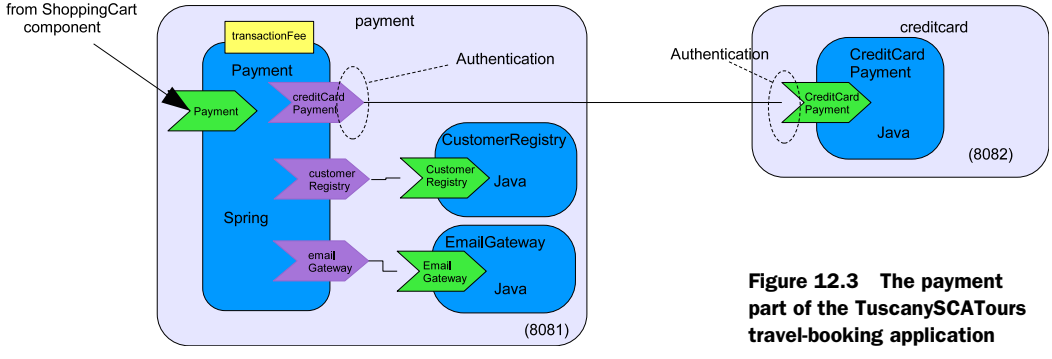


Figure 12.3 The payment part of the TuscanySCATours travel-booking application

The `payment` reference of the ShoppingCart component in figure 12.2 is connected, over Web Services, to the Payment service of the Payment component in figure 12.3.

Remember that these diagrams, and the diagrams in the rest of this chapter, show only the composites involved in the application. Running the sample requires the contributions that contain these composites, along with a number of other contributions. The full list of sample contributions required is shown in listing 12.1.

In the following subsections we'll provide an overview of each composite in turn. The name in parentheses in each title is the name of the contribution being discussed. We'll start by looking at how the web page is built.

12.2.1 The application user interface (fullapp-ui)

The Web 2.0–based TuscanySCATours user interface is displayed using the SCATours-UserInterface SCA component. This in turn is implemented using implementation.widget. This handles the various parts of the application user interface, which was shown in figure 12.1.

The user interface won't win any prizes for interface design, but it provides enough features for us to demonstrate components behind the interface doing something interesting. If you look closer at the fullapp-ui contribution, as shown in figure 12.4, you'll see two components defined.

Figure 12.4 shows that the SCAToursUserInterface component is wired to the three services provided by the SCATours component. The contents of the fullapp-ui.composite file are shown here.

Listing 12.2 The fullapp-ui.composite file from the `fullapp-ui` contribution

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
           name="fullap-ui">
```

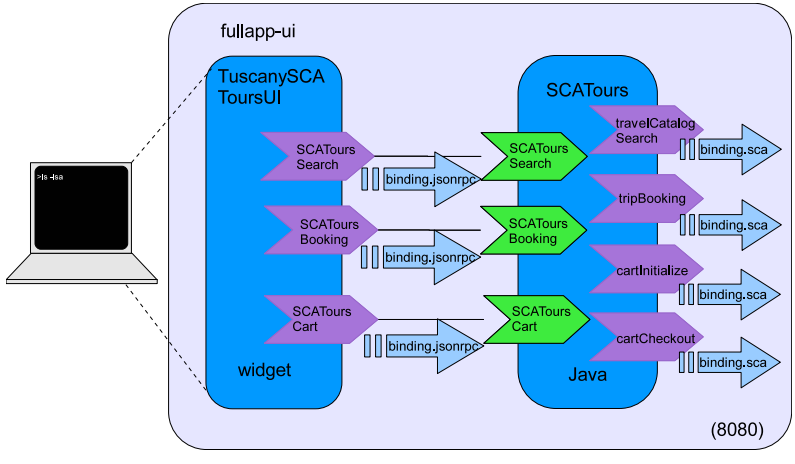


Figure 12.4 The `fullapp-ui` contribution provides the user interface for the TuscanySCATours application.

```
    <component name="SCAToursUserInterface">
        <tuscany:implementation.widget location="scatours.html"/>
        <service name="Widget">
            <tuscany:binding.http uri="/scatours"/>
        </service>
        <reference name="scaToursCatalog"
                   target="SCATours/SCAToursSearch">
            <tuscany:binding.jsonrpc/>
        </reference>
        <reference name="scaToursBooking"
                   target="SCATours/SCAToursBooking">
            <tuscany:binding.jsonrpc/>
        </reference>
        <reference name="scaToursCart" target="SCATours/SCAToursCart">
            <tuscany:binding.jsonrpc/>
        </reference>
    </component>

    <component name="SCATours">
        <implementation.java
            class="com.tuscanyscatours.impl.SCAToursImpl"/>
        <service name="SCAToursSearch">
            <tuscany:binding.jsonrpc/>
        </service>
        <service name="SCAToursBooking">
            <tuscany:binding.jsonrpc/>
        </service>
        <service name="SCAToursCart">
            <tuscany:binding.jsonrpc/>
        </service>
        <reference name="travelCatalogSearch"
                   target="TravelCatalog/TravelCatalogSearch"/>
        <reference name="tripBooking" target="TripBooking"/>
        <reference name="cartInitialize"
                   target="ShoppingCart/CartInitialize"/>
        <reference name="cartCheckout" target="ShoppingCart/CartCheckout"/>
    </component>
</composite>
```

2 jsonrpc binding

3 Java implementation

The SCAToursUserInterface component is implemented using an implementation.widget type 1. As you learned in chapter 8, the SCATours component runs as JavaScript in the user's browser. JavaScript supports a limited number of reference bindings, one of which is `binding.jsonrpc` 2, which we're using to communicate with the various services of the SCATours component.

The SCATours component allows us to control how the user interface communicates with the rest of the backend components. SCATours is a normal SCA component running on the server and is implemented using implementation.java 3. It provides services to the browser over the `binding.jsonrpc`. Because it's running inside the Tuscany runtime, we're free to configure its references with any of the bindings that Tuscany supports. In this case it uses the default binding to communicate with the TravelCatalog, TripBooking, and ShoppingCart components.

Let's look at the services that the SCATours component is connected to.

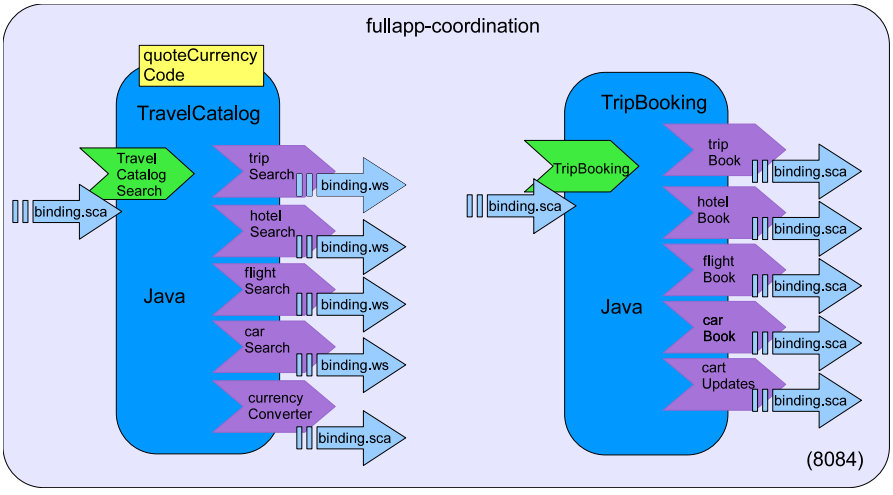


Figure 12.5 The TravelCatalog and TripBooking components coordinate communication with the services provided by the TuscanySCATours company's business partners.

12.2.2 Coordinating the application (fullapp-coordination)

TuscanySCATours uses a series of partner organizations to provide hotels, flights, cars, and even prepackaged trips. The user will initially search for available trips. The user may then select some of the trips that have been identified and proceed to book them. Both of these actions require interaction with TuscanySCATours partner organizations.

To coordinate this activity, the TuscanySCATours application has a fullapp-coordination contribution that acts as a typical middle tier, providing coordination between the frontend user interface and the backend services that provide search and booking functions. It does this using two components, TravelCatalog and TripBooking, as shown in figure 12.5.

The TravelCatalog component manages the process of searching for trip items, whereas the TripBooking component handles bookings on a user's behalf. The TravelCatalog component provides a `TravelCatalogSearch` interface with a `search` operation, which takes a `TripLeg` argument, as follows:

```
@Remotable
public interface TravelCatalogSearch {
    TripItem[] search(TripLeg tripLeg);
}
```

The `TripLeg` argument holds the information gathered from the user interface regarding the details of the leg of the trip the user is searching for, for example, the start and end dates and details of where the user wants to travel to and from. The `TripItem` array that's returned has one entry for each item found during the search. An item can be a trip, hotel, flight, or car.

This component determines what TuscanySCATours has to offer when a search is performed by the user. During the processing of the search, the TravelCatalog uses its

`tripSearch`, `hotelSearch`, `flightSearch`, and `carSearch` references to search for trips, hotels, flights, and cars. The component services that are wired to these references represent gateways to the TuscanySCATours company's business partner services that provide the actual trips, hotels, flights, and cars.

To ensure that the search happens as quickly as possible, the TravelCatalog component calls out to all the referenced partner components in parallel using the one-way interaction pattern. You'll notice a delay of a few seconds when you perform a search because we added artificial delays in the partner services.

Once the results are returned using the callback interaction pattern, the quoted prices are converted into the currency specified via the quoteCurrencyCode component using the currencyConverter service. The results are then returned to the user interface.

The TripBooking component has a similarly simple interface, as follows:

```
@Remotable
public interface TripBooking {
    TripItem bookTrip(String cartId, TripItem trip);
}
```

When the user selects and then books an item, the `bookTrip` operation is used to provisionally book the item with the appropriate partner and then add the item to the shopping cart. The `cartId` parameter here represents the user's session. The user interface creates a new cart when the user connects and uses the `cartId` to identify the user as required. This isn't a sophisticated user-tracking system but is sufficient for this sample.

Next we'll look at the SCA services that represent the TuscanySCATours company's business partners. The TravelCatalog and TripBooking components use these services to search for and book trip items, respectively.

12.2.3 Partner services (fullapp-packagedtrip and bespoketrip)

The TripPartner, HotelPartner, FlightPartner, and CarPartner components are all owned and run by the different departments of the TuscanySCATours company. Each department is responsible for managing the relationship with a different type of supplier. The services are organized to reflect this responsibility, as shown in figure 12.6.

In figure 12.6 we've identified departments that are responsible for components in the application by adding appropriately labeled boxes. These boxes don't represent any particular SCA syntax and are just for the purpose of this explanation. Using SCA, each department can build and package its part of the application in isolation. Only interface descriptions need be shared.

For example, the Offerings Department owns the TravelCatalog component and packages it alongside the TripBooking component in the fullapp-coordination contribution. The Packaged Trip Department owns the TripPartner component and packages it in the fullapp-packagedtrip contribution. Only the Search and Book interface descriptions are shared between the two departments.

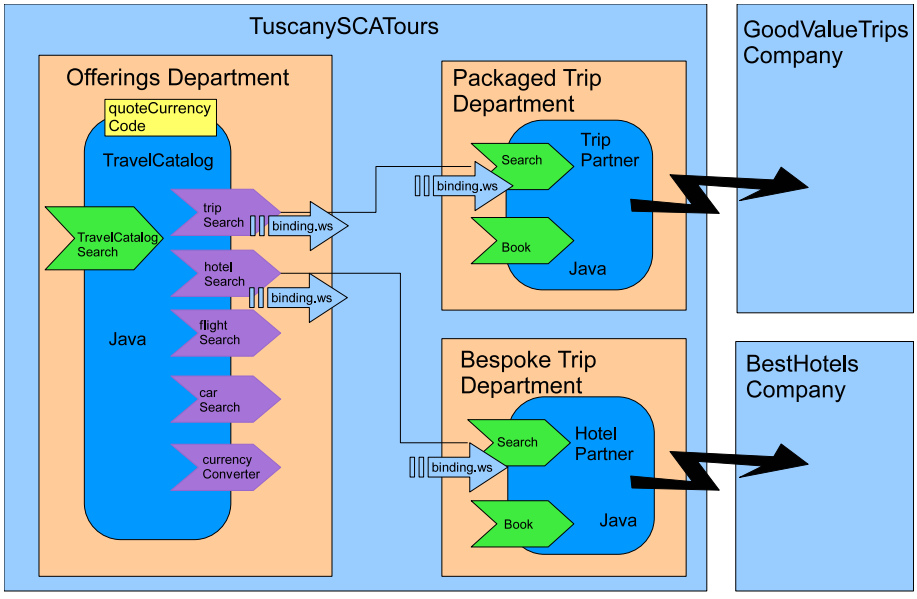


Figure 12.6 The components in the TuscanySCATours application are aligned with the kinds of departments that could be part of a travel-booking company.

The individual partner components are free to use whatever means are appropriate to communicate with the partner companies. This communication is shown by the jagged arrows in figure 12.6. It could involve telephone, fax, email, or even direct service calls using SCA references with bindings such as the Web Services binding.

In our sample application, for simplicity, no references are provided, and the partner components have travel information hardcoded internally.

Let's look a little more closely at the `Search` and `Book` interfaces used by the partner services and review the way that the SCA one-way and callback interaction patterns have been used.

THE SEARCH INTERFACE

The search-related references of the TravelCatalog component use the same `Search` interface as shown in the next code snippet. The different departments have agreed to use the same interface, and this interface has been constructed so that all of these departments' search services can operate in parallel.

```
@Remotable
@Callback(SearchCallback.class)
public interface Search {
    TripItem[] searchSynch(TripLeg tripLeg);
    @OneWay
    void searchAsynch(TripLeg tripLeg);
    int getPercentComplete();
}
```

The `Search` interface provides operations for searching synchronously, `searchSynch`, and for searching asynchronously, `searchAsynch`. To support asynchronous searching, the `Search` service interface is defined with a callback interface called `SearchCallback`. The TripCatalog component uses the `searchAsynch` operation on the TripPartner, HotelPartner, FlightPartner, and CarPartner components to initiate four searches in parallel.

When the sample application is run, all searches performed will take several seconds to complete. This is because each of the partner services contains an artificial delay to simulate a real search being performed across trip-, hotel-, flight-, or car-booking information. For example, look at the implementation of the Flight component's `searchAsynch` operation:

```
public void searchAsynch(TripLeg tripLeg) {

    while ( percentComplete < 100 ){
        try {
            Thread.sleep(50);
        } catch(Exception ex){}
        percentComplete = percentComplete + 10;
        searchCallback.setPercentComplete(componentName, percentComplete);
    }
    searchCallback.searchResults(searchSynch(tripLeg));
}
```

Note the call to `Thread.sleep`, which introduces a delay into the processing to simulate search processing. This operation makes calls to a callback service. The TravelCatalog component implements the `SearchCallback` interface. The callback interface is shown next:

```
@Remotable
public interface SearchCallback {
    void searchResults(TripItem[] items);
    void setPercentComplete(String searchComponent, int percentComplete);
}
```

The `searchResults` operation is called by each partner component when it has finished searching. This is how the results are returned to the TravelCatalog component. The configuration of the callback pattern is evident if you look at the composite descriptions of the TravelCatalog component and one of the partner components, such as HotelPartner. First, let's look at the TravelCatalog component, which is defined in the fullapp-coordination.composite file:

```
<component name="TravelCatalog">
    <implementation.java class=
        "com.tuscanyscatours.travelcatalog.impl.TravelCatalogImpl"/>
    <service name="TravelCatalogSearch"/>
    <reference name="hotelSearch">
        <binding.ws uri="http://localhost:8086/Hotel/Search"/>
        <callback>
```

```
            <binding.ws name="callback"
                uri="http://localhost:8084/Hotel/SearchCallback"/>
        </callback>
    </reference>
</component>
```

The `hotelSearch` reference defines a forward `binding.ws` and a callback `binding.ws`. This reference will send messages out to http://localhost:8086/Hotel/Search and listen for messages arriving at http://localhost:8084/Hotel/SearchCallback.

If you now look at the definition of the HotelPartner component, which is defined in the fullapp-bespoketrip.composite file, you can see how the service is configured to support these callbacks:

```
<component name="HotelPartner">
    <implementation.java class=
        "com.tuscanyscatours.hotel.impl.HotelImpl"/>
    <service name="Search">
        <binding.ws name="searchws"
            uri="http://localhost:8086/Hotel/Search"/>
        <callback>
            <binding.ws/>
        </callback>
    </service>
    <service name="Book"/>
</component>
```

Using the same style of configuration as with the previous reference, the Search service is configured to listen to messages arriving at http://localhost:8086/Hotel/Search. In this case the Web Services binding assigned to the callback has no URI configuration. This means callbacks will be directed to the callback service of the reference that sent the forward call.

The results from the partner services will arrive back at the TravelCatalog component at various times as each search process completes. Once the TravelCatalog component has received results from all of the partner components it has called, it collates all of the results, converts the quoted prices into the correct currency using the CurrencyConverter component, and returns the results to the user interface via the SCA-Tours component.

This asynchronous interaction pattern using callbacks is described in detail in chapter 4.

THE BOOK INTERFACE

The `Book` interface is similarly straightforward and provides a single operation for booking trips, as follows:

```
@Remotable
public interface Book {
    String book(TripItem tripItem);
}
```

The partner components implement this interface, and the TripBooking component uses this interface to notify the partners when the user has booked a trip. In the current

application the partner components don't do anything interesting with bookings, but this is where the TuscanySCATours company would ensure that the trips that the user wants will be reserved for them.

As we mentioned when we talked about the `Search` interface, the TravelCatalog component uses the CurrencyConverter component to ensure that quoted prices appear in the correct currency. We'll look at that component next.

12.2.4 Currency conversion (fullapp-currency)

The CurrencyConverter component is a simple component that provides a single service with the following interface:

```
@Remotable
public interface CurrencyConverter {
    double getExchangeRate(String fromCurrencyCode,
                           String toCurrencyCode);
    double convert(String fromCurrencyCode,
                   String toCurrencyCode,
                   double amount);
}
```

The TravelCatalog component calls the currency converter, via a reference called `currencyConverter`, in the following way:

```
tripItem.setPrice(currencyConverter.convert(
    tripItem.getCurrency(), quoteCurrencyCode, tripItem.getPrice()));
```

The currency that the trip item is originally quoted in is converted into the currency that the TravelCatalog is configured to quote prices in. This configuration takes place via an SCA property defined in the following way:

```
@Property
public String quoteCurrencyCode = "USD";
```

The default is USD, but the value is configured in the composite description of the TravelCatalog component as follows:

```
<component name="TravelCatalog">
    <property name="quoteCurrencyCode">GBP</property>
</component>
```

The implementation, service, and reference elements have been left out in this case for clarity.

The next part of the application deals with how booked trip items are accumulated so that they may eventually be purchased. Let's look at the shopping cart.

12.2.5 Constructing trips (fullapp-shoppingcart)

The travel-booking application stores the items that the user selects to buy. This state is managed by the ShoppingCart component and the CartStore component to which it's wired. The fullapp-shoppingcart contribution contains the composite file and references the shoppingcart contribution, which contains the interfaces and implementations.

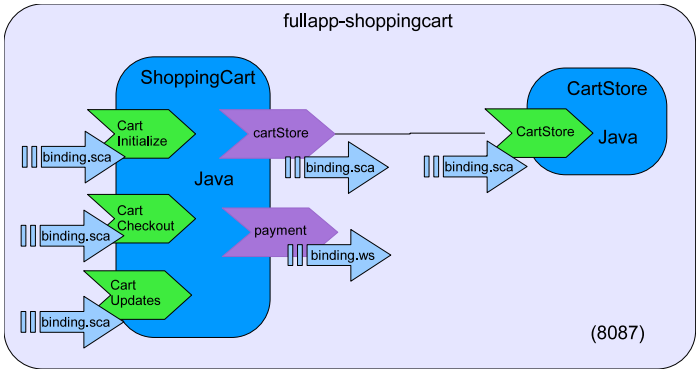


Figure 12.7 The Shopping Cart composite containing the ShoppingCart component and the CartStore component that stores items added to the shopping cart

When the user uses the travel-booking application, a cart ID is created that identifies their shopping cart until checkout. At checkout, the shopping cart store is emptied and removed, and a new ID is generated. The two components are shown wired together in figure 12.7.

In figure 12.7 you can see that the ShoppingCart component is wired to the CartStore component via the `cartStore` reference. Each instance of the CartStore component represents a shopping cart. In the current application the contents of the cart store are held in memory, but in a more robust implementation the contents of a cart store are likely to be saved to disc on each update.

As described back in chapter 4, the CartStore component implements a conversational interface, as shown in the following listing.

Listing 12.3 The conversational `CartStore` interface

```
@Remotable
@Conversational
public interface CartStore{
    void addTrip(TripItem trip);
    void removeTrip(TripItem trip);
    TripItem[] getTrips();
    @EndsConversation
    void reset();
}
```

A new instance of the CartStore component is generated when the `addTrip` operation is called for a new conversation ID. Tuscany generates conversation IDs automatically. This works well if you don't need to refer to the ID in your application code. We choose to generate conversation IDs manually, inside the `newCart` operation of the ShoppingCart component, so that we can use the ID elsewhere in the code, as follows.

Listing 12.4 The mechanics of generating a cart ID in the ShoppingCart component

```
@Reference
protected CartStore cartStore;
@Context
protected ComponentContext componentContext;
```

```
private static Map<String, CartStore> cartStores =
    new HashMap<String, CartStore>();

public String newCart(){
    String cartId = UUID.randomUUID().toString();
    ServiceReference<CartStore> cartStore =
        componentContext.getServiceReference(CartStore.class,
            "cartStore");
    cartStore.setConversationID(cartId);
    cartStores.put(cartId, cartStore.getService());
    return cartId;
}
```

1 Active CartStore conversations

2 Generate conversation ID

In listing 12.4 you can see that the conversation ID is generated 2 and returned as a result of the `newCart` call. From there the ID is passed back to the JavaScript running in the browser. In turn the JavaScript can pass it back in with any service calls that update the shopping cart. This is how the shopping cart for one user is distinguished from the shopping cart for another.

In SCA, it's not possible to retrieve an automatic conversation ID before a call has been made to a conversational service. When a call arrives at the `newCart` operation, no calls have yet been made to the cart store, and so we'll generate the conversation ID ourselves.

We hold a list of references to CartStore component instances in the `cartStores` map 1. This allows the right CartStore component reference to be found in subsequent calls given the incoming `cartId`.

The mechanism by which a CartStore component instance is subsequently found deserves a closer look. Remember that each instance of the CartStore component represents the state of a separate shopping cart. When a new cart is requested, the `newCart` operation creates the conversation ID (called `cartId`), gets a new service reference from the component context, and sets the conversation ID on the reference. The reference is stored away in the `cartStores` map for later use. When a request comes in later to add a trip item to the shopping cart, the `cartStores` reference map is used in the following way:

```
public void addTrip(String cartId, TripItem trip) {
    cartStores.get(cartId).addTrip(trip);
}
```

Looking back at listing 12.3, you can see that the `reset` operation of the `CartStore` interface is marked with the `@EndsConversation` annotation, which is used when the shopping cart has finished with this cart store and the state that it holds can be removed. This operation is called when the user clicks the checkout button. This calls the `checkout` operation on the ShoppingCart, which, after processing the payment, removes the cart store, as shown here.

Listing 12.5 The ShoppingCart component checkout operation

```
public void checkout(String cartId, String customerName){
    String customerId = customerName;
    float amount = (float)0.0;
```

```
    TripItem[] trips = getTrips(cartId);
    for (TripItem trip : trips){
        if (trip.getType().equals(TripItem.TRIP)){
            amount += trip.getPrice();
        } else {
            for (TripItem tripItem : trip.getTripItems()){
                amount += tripItem.getPrice();
            }
        }
    }
    payment.makePaymentMember(customerId, amount)
    cartStores.get(cartId).reset();
    cartStores.remove(cartId);
}
```

The main body of code in the `checkout` operation gets the trips from the cart store using the `cartId` as a key. It totals up the price for all the items in the cart store for this `cartId` and then calls the Payment component to make a payment on behalf of the user. Finally, the cart store is reset, which ends the conversation that's ongoing between the ShoppingCart and CartStore components for this `cartId`. After this, the cart store reference is removed from the `cartStores` map.

If the user continues shopping, then a new cart is created, which results in a new conversation ID being generated, and the process starts again. Looking back at listing 12.5, you can see that during checkout processing the ShoppingCart component uses a reference called `payment` in order to take a payment from the user. Let's look at that now.

12.2.6 Payment processing (payment and creditcard)

The payment-related components have been used numerous times already in this book to demonstrate a variety of implementation types, bindings, and databindings. In the full application we're using the `payment-spring-policy` contribution to process the payments and the `creditcard-payment-jaxb-policy` contribution to process customer credit cards.

Payment processing isn't the core business of the TuscanySCATours company, and so we've implemented a Payment component that orchestrates the payment process but offloads the real work to other components, as shown in figure 12.8.

A CreditCardPayment component handles the processing of customer credit cards. We've included a CustomerRegistry component that retrieves customer information, such as stored credit card details. There's also an EmailGateway component, which, in real life, would send email back to the user confirming that a payment has successfully been taken.

Looking inside the implementations of the components here, you'll see that they aren't proficient from a payment-processing point of view. They don't do any real payment processing or send any real emails. They do, however, demonstrate some interesting features of Tuscany.

First, as described in chapter 6, we're mixing implementation types. Here the Payment component is implemented as a Spring context. The Payment component

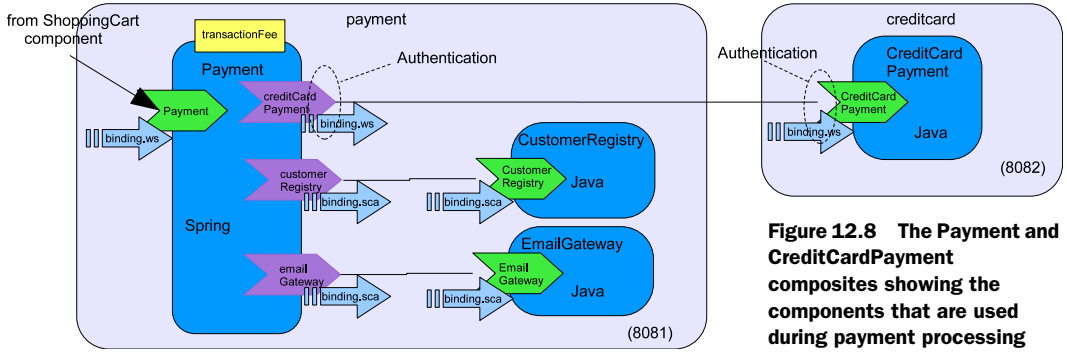


Figure 12.8 The Payment and CreditCardPayment composites showing the components that are used during payment processing

references out to a CustomerRegistry over the SCA binding and to the CreditCard component over the Web Services binding. The Spring implementation of the Payment component isn't aware of how these other components are implemented. SCA allows us to combine multiple implementation technologies in a single composite application.

The CreditCard component could be implemented by another company, perhaps using non-SCA technologies. Here we've shown it implemented using SCA because it made it quick to develop. Because the web service communication between the Payment and CreditCardPayment components may pass between companies, we've added the `authentication` intent to the `creditCardPayment` reference of the Payment component as follows:

```
<component name="Payment">
    <implementation.spring location="Payment-context.xml"/>
    <service name="Payment">
        <binding.ws uri="http://localhost:8081/Payment"/>
    </service>
    <reference name="creditCardPaymentReference" requires="authentication">
        <binding.ws uri="http://localhost:8082/CreditCardPayment"/>
    </reference>
    <reference name="emailGateway" target="EmailGateway"/>
    <reference name="customerRegistry" target="CustomerRegistry"/>
    <property name="transactionFee">1.23</property>
</component>
```

The `CreditCard` interface of the CreditCardPayment component is also configured with a Web Services binding and `authentication` intent:

```
<component name="CreditCardPayment">
    <implementation.java class="com.tuscanyscatours.payment.
    ➥ creditcard.impl.CreditCardPaymentImpl" />
    <service name="CreditCardPayment">
        <interface.wsdl interface="http://www.tuscanyscatours.com/
        ➥ CreditCardPayment/#wsdl.interface(CreditCardPayment)" />
        <binding.ws uri="http://localhost:8082/CreditCardPayment"
            requires="authentication"/>
        <binding.sca/>
    </service>
</component>
```

Looking back at listing 12.1, note that we've used the `payment-spring-policy` and `creditcard-payment-jaxb-policy` contributions directly without referencing them from another contribution, such as a `fullapp-payment` contribution, as we've done in other parts of the application. This has the benefit that the configuration required to process payments is entirely contained in these contributions. The drawback is that these contributions specify a deployable composite, which makes them more difficult to reuse in other composites. This is because the deployable composite provided will be deployed regardless of whether you write a new composite that uses the payment components in a slightly different way.

The payment processing shows the use of a different implementation type and includes a `policy` intent. The ShoppingCart component calls the Payment service `makePayment` operation when the user checks out, so payment can be taken for the items in the shopping cart. The full application assumes that the user has already registered with TuscanySCATours using a separate web page, and so the Payment component uses the CustomerRegistry component to retrieve the preregistered customer details, including their credit card number. The Payment component then calls the `authorize` operation on the CreditCardPayment service to authorize the total amount against the credit card number. The EmailGateway component is then used to pretend to send an email detailing whether the payment was taken successfully or not. Once payment has been taken and an email has been sent, the shopping cart is emptied for this user, and the process can start again.

That brings us to the end of our walkthrough of the components used in the TuscanySCATours application. We've run the application in a single node using the launchers/fullapp module, so let's now try running the TuscanySCATours application in a distributed domain.

12.3 The travel-booking application in a distributed domain

We started this chapter by running the application in a single JVM using the `FullAppLauncher` class. This is a useful configuration because it allows us to run and debug the application easily when we're assembling components.

Now let's try the application running across several distributed nodes. You'll need to do this if different parts of the application are to be run on different physical machines. This could be because the machines are owned by the parts of the organization that own the components or for performance reasons or even for security reasons. If you don't need to run SCA applications across multiple JVMs yet, you can safely skip this section and go straight to section 12.4.

Nothing in the components, composites, or contributions changes when we run across multiple JVMs. We're just going to assign the composites to Tuscany nodes running separately.

To help us do this, we're going to use Tuscany's domain manager. The domain manager is a separate program that manages contributions in a domain. Individual nodes running in the domain communicate with the domain manager to retrieve the

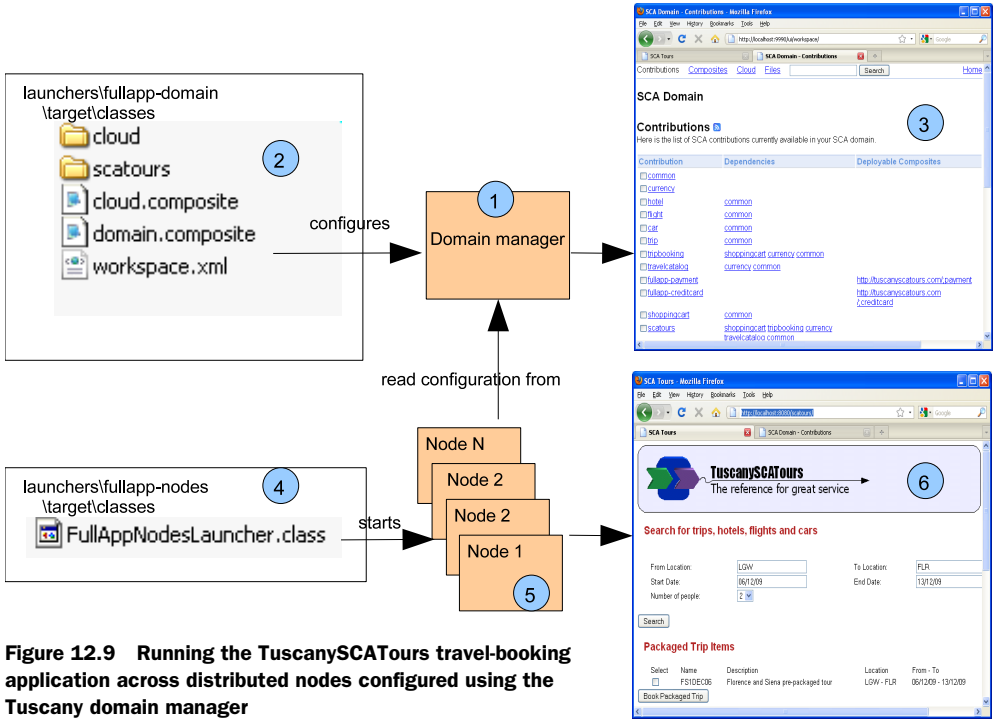


Figure 12.9 Running the TuscanySCATours travel-booking application across distributed nodes configured using the Tuscany domain manager

details of the part of the distributed SCA application that's to run. Chapter 3 gives a good overview of the features of the domain manager and how to use it, so we won't repeat that here.

For the TuscanySCATours application, the domain manager is started using the launchers/fullapp-domain module. Once we've started the domain manager, we'll start several nodes, one for each composite in the application, which will each read its configuration from the domain manager across the network. All the separate nodes are started using the launchers/fullapp-nodes module. Figure 12.9 gives an overview of the running sample.

From chapter 3 you may remember that the domain manager, box 1, is configured by a set of files shown in box 2. The details of the contents of the files shown in box 2 are described in chapter 11.

You need to set up your environment before running the sample, following the instructions in appendix A. Then, assuming you have the source code distribution of the travel-booking application, enter the following:

```
cd launchers/fullapp-domain
ant run
```

The `ant` command runs the `run` target in the build.xml Ant script. This in turn runs the Java launcher program, which, in this case, is called `FullAppDomainLauncher`.

When run, the domain manager will first read the names of all of the application contributions from the workspace.xml file. It will then read, from the domain.composite file, the names of all of the composites from these contributions that are to be deployed into the domain. Finally, it reads the configuration of all of the nodes that it's expecting to be started from the cloud.composite file and from the individual node configuration files in the cloud subdirectory. Once it's started, you'll see the following message:

```
[java] INFO: Press 'q' to quit, 'r' to restart.
```

If you point your browser at the following URL, you'll be able to see the domain manager's management interface, similar to what's shown in box 3 of figure 12.9: http://localhost:9990/ui/workspace.

As described in chapter 3, this domain manager user interface allows you to adjust the configuration of the domain manager. You can also start nodes from this user interface, but in this case we're going to use a program to start all the nodes for us.

The nodes, box 5, are started by running the `FullAppNodesLauncher` class in the launchers/fullapp-nodes module, box 4. From the travel-booking application source code distribution, you can run it by entering the following:

```
cd launchers/fullapp-nodes
ant run
```

The `FullAppNodesLauncher` starts several nodes, one for each composite in the application. Using this launcher, all of the nodes will be started in the same JVM.

We could have started each node in a separate JVM, and the effect would have been the same. This means that each of the nodes in box 5 could be running on a separate computer with little extra effort.

This is how the node that runs the creditcard composite is started:

```
SCANode nodeCreditcard = SCANodeFactory.newInstance().
    createSCANodeFromURL("http://localhost:9990/node-config/creditcard");
nodeCreditcard.start();
```

The node is configured with the URL of the domain manager extended with a unique string that identifies this node's configuration. The domain manager, box 1, makes the configuration available at this URL based on the configuration it's been provided with, box 2. Once all the nodes are started, you'll see the following:

```
[java] Point your browser at - http://localhost:8080/scatours/
[java] Nodes started - Press enter to shutdown.
```

If you point your browser at the URL described in this message, you should see the TuscanySCATours travel application front page, box 6, that you saw back in figure 12.1.

We've finished running the application now, but before we close out this chapter, let's talk through some of the tricks and tips you learned while developing the application.

12.4 Hints and tips for building composite applications

Here we'll present some hints and tips that may be useful to you as you build your applications. This isn't an exhaustive collection of best practices for SCA or Tuscany, and we'd be glad to hear from you on the Tuscany mailing list if you have any hints or tips of your own to share.

12.4.1 Prototyping and then filling out

The Tuscany SCA Java runtime is helpful in the way that, with no extra configuration, it will automatically start up components and the bindings that are used to wire components together.

We started to build the application using a top-down approach by first sketching out the composites and components that served the basic business purpose. Once the basic building blocks were clear, we manually created simple Java implementations for the components. Finally, we described them in composite files and wired them together.

These artifacts were easy to combine in a single contribution constructed using a Maven module or Eclipse project along with some simple tests to exercise the component services.

At this stage, after little time and effort, we were able to run up the components and experiment with their construction and wiring. A good example of this is the set of partner services and the TravelCatalog component that uses them. The interactions between these components are relatively complex in that callbacks and remote bindings are involved. Building this set of components didn't require any specific configuration of the underlying protocols; it was just a matter of writing the composite configuration, constructing the simple Java component implementations, and running the resulting composites.

This approach suits test-driven development and allows for rapid prototyping, after which the application model can be evolved to bring in new implementation types, new bindings, real and functional implementations, and reorganization into separate contributions.

12.4.2 Application organization

We previously pointed out in this chapter that the full application is made up of a combination of contributions that contain component implementation assets and others that exploit the assets contained in other contributions. This turned out to be quite interesting. Figure 12.10 shows the payment-java contribution, which contains everything required to run the Payment component.

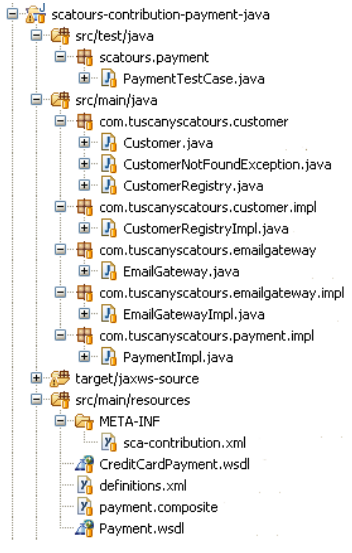


Figure 12.10 The layout of the self-contained `payment-java` contribution

Notice that this combines the payment.composite file along with all of the assets that the composite requires to run, such as the Java classes and WSDL interface definitions. This is convenient for testing, but it means we'll have to break open the contribution and change the composite file if we want to change the wiring or add another component.

We found it convenient to create a contribution to hold the composite file, which, in turn, references a separate contribution holding the implementation assets. This exploits a contribution's ability to export Java packages and attribute namespaces for others to use. For example, the common contribution contains Java interfaces, which are shared by many of the other contributions in the application, and exports the `com.tuscanyscatours.common` package for other contributions to import.

All the contributions whose name starts with fullapp- use this approach. Figure 12.11 shows an example, the fullapp-bespoketrip contribution.

This then references the assets of the trip, hotel, flight, and car contributions, which hold the Java assets required to run the components defined in the fullapp-bespoketrip.composite file. Looking at the sca-contribution.xml file, you can see these relationships explicitly stated:

Figure 12.11 The layout of the fullapp-bespoketrip contribution

```
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
              xmlns:scatours="http://tuscanyscatours.com/">
    <deployable composite="scatours:bespoketrip" />
    <import.java package="com.tuscanyscatours.common" />
    <import.java package="com.tuscanyscatours.hotel" />
    <import.java package="com.tuscanyscatours.hotel.impl" />
    <import.java package="com.tuscanyscatours.flight.impl" />
    <import.java package="com.tuscanyscatours.car.impl" />
</contribution>
```

When it comes time to update the composite file, we can edit the existing composite file and rebuild the `fullapp-bespoketrip` contribution without changing any of the referenced contributions. More likely though, we'll create a `fullapp-bespoketrip-v2` contribution to hold the changes, but this doesn't involve copying any other assets because these remain in their separate contributions.

The object here is to be in a position to react to changes in the business process without needing to reconstruct all of the prebuilt and tested contributions.

None of this means that composite files won't be placed inside contributions alongside the assets that they use. One good reason for doing this is contribution testing, where we test all of the components that a contribution contains in isolation before they're used in a wide composite application. We've often done this. Looking back at the `payment-java` contribution in figure 12.10, you see a test package containing a JUnit4 test case. In Tuscany this can easily reference a local composite without declaring the composite as deployable for the contribution as a whole.

We should note that we've often made copies of contributions on purpose when writing this book to make the samples easier to follow. When concentrating on a particular aspect, such as policy or databinding, all the assets are available in a single contribution and aren't spread about. But this approach would lead to a lot of unnecessary copying in real applications; combining shared assets into common contributions, as discussed here, reduces duplication.

12.4.3 Developing contributions in a team

Continuing from the previous point, the partitioning of an application into a number of contributions allows the work to be partitioned and assigned to different members of a team or even to different teams.

In the sample application is a contribution called `common` that's shared by most of the other contributions. This contains a small number of assets that most other contributions need. Other than the `common` contribution, the rest of the contributions are relatively standalone and can be implemented in isolation. The `service` interfaces provide the point of control between the contributions.

By combining this thought with the previous idea of prototyping contributions, it's easy for a high-level design team to mock up the system and then farm out the prototype contributions to separate teams who will write a fully functional implementation.

Contributions that you intend to share out across the team can be checked into a version control system or added to a registry or shared directory to allow physical access to them. When developing this application we hosted the source code for the modules that are used to generate the shared contributions in a version control system. People wanting to use them checked them out at the required version and built the contributions themselves.

12.4.4 Testing contributions in a single VM

In this chapter we've presented the travel-booking application, running it first in a single VM followed by running it across a distributed set of nodes. The flexibility and automation provided by the Tuscany runtime support these two modes of operation while requiring no changes to the application in order to switch between the two.

Starting with the single node of operation will help get applications up and running quickly. Start with single contributions and take a test-driven approach to building components and the service interfaces they provide. Combining components then allows you to build more complex compositions and work toward the full application. It's easy to run in this mode regardless of whether you're developing applications using command-line tools or from an IDE such as Eclipse. It's easy to start, stop, and debug the single-node configuration.

Once you have code running to your satisfaction in a single node, you can run it across multiple nodes to distribute the workload if required. You can do this by using

the Tuscany domain manager to configure your nodes, as we've done in this chapter when running the TuscanySCATours application.

If you don't want to run a separate domain manager process to manage the domain, you can easily achieve a similar affect by manually configuring SCA bindings in your application composite files to represent the remote communications between the nodes running separate parts of the application.

12.4.5 Top-down and bottom-up development

Tuscany allows for both top-down and bottom-up development. In the bottom-up approach you build the component implementation and generate the service interface descriptions from there. In the top-down approach it's the other way round. You build the service interface descriptions manually first and then build the service implementations based on these interfaces.

Which style you choose depends on the situation. We've found that the bottom-up approach works well when prototyping a single component. A component prototype can be evolved incrementally until a satisfactory service interface emerges. In this case Tuscany is able to generate WSDL service interface descriptions automatically if `binding.ws` is included on the services. To get the WSDL, add `?wsdl` to the end of the service endpoint URL and point a browser at the resulting URL.

The top-down approach will come into play if there's an existing service interface to work with. This happened in our application where components share an interface; for example, the Payment component calls the CreditCardPayment component over the Web Services binding. In this case we imagined that the CreditCardPayment component would be a service provided by a different company, and so we generated WSDL for the service interface. The TuscanySCATours company then generates a service interface from the WSDL in an appropriate programming language, in our case the Java language. In this case we used the wsimport tool that's provided with the Java JDK to read in the CreditCardPayment WSDL and generate a Java interface. We could've done this from the command line, but we chose to configure the Maven build for the Payment contribution module to run wsimport for us. You can see this configuration if you look in the pom.xml file that's at the root of the `payment-java` contribution.

Because these are general patterns that you're likely to encounter, you may find the contents of the pom.xml (if you're a Maven fan) or build.xml (if you're an Ant fan) file useful when creating your own contribution modules.

12.4.6 Recursive composition

We'll close this discussion of hints and tips with a word about recursive composites. *Recursive composition* is a term that describes the use of an SCA composite to implement an SCA component. In effect, composites are nested recursively one inside another. You can see when a composite is providing the implementation of a component

because the definition of the component in the composite file will include the `implementation.composite` element.

We haven't used any recursive composites in the TuscanySCATours application. This doesn't mean that the technique isn't useful for describing reusable functions. It's just that the opportunity didn't arise in this small application.

12.4.7 SCA and versioning

The SCA specifications don't discuss how to version contributions, or any other SCA artifacts, and the Tuscany SCA Java runtime doesn't provide any specific support for versioning.

From a development point of view, we relied on a version control system to version the source for all of the contribution modules in the sample application. This allowed us to collaboratively develop the contributions and keep track of all the incremental changes.

The simplest thing to do from a runtime point of view is to make sure that you give your contributions sensible names. Because there is no metadata associated with a contribution that indicates its version, a simple version number in the contribution archive name, as you would with a JAR file, works well. You'll note that we didn't do this in the sample to keep the contribution names as short as possible. Maven will generate target names with version numbers automatically, but we turned off this feature using the following section of each contribution's pom.xml file:

```
<build>
    <finalName>${artifactId}</finalName>
    ...
</build>
```

We've been demonstrating the sample application, running it in a single SCA domain, and there's no support in the domain for managing different versions of a given contribution at the same time. To do this you'll need to run the different versions in separate contributions.

That's it for this section, so let's now move on and wrap up this chapter.

12.5 Summary

Building an enterprise application will almost certainly involve many different people building many different cooperating services. SCA and Tuscany have been designed to help you build these kinds of applications and to control the complexity that naturally occurs. Here we've shown you how SCA and Tuscany can be used to construct an application that relies on multiple interconnected components.

The majority of this book so far has involved looking at specific parts of SCA and Tuscany using separate elements of our sample TuscanySCATours travel-booking application. In this chapter we've brought those parts together and shown you how easy it is to assemble a complete running application. First, we ran the application in a single

node, a configuration that makes testing and incremental development straightforward. Following this, we showed that by restarting the application using a suitably configured Tuscany domain manager, we're able to run the application using a distributed infrastructure without changing the application itself.

Beyond the idea of building a composite application out of assembled components, we've also described a few other ways that SCA and Tuscany can help speed your development process. This is based on leveraging the SCA model of assembled components in order to distribute work between a team of developers and define an incremental implementation strategy. The power of SCA is the model that it presents of an assembled application. Don't be afraid to exploit it.

Now we've covered all aspects of developing, deploying, and managing applications using Tuscany SCA. We're going to switch gears in the next part of the book and look at how the Tuscany runtime is architected and how we can create extensions to add additional capabilities such as new implementation types.

Part 4

Exploring the Tuscany runtime

You've now learned how to use Tuscany to design, develop, deploy, and run SCA applications. We hope you've enjoyed the SCA experience of writing service components and composing them into applications without having to struggle with infrastructure plumbing. How does this plumbing magic happen? Can you extend it? Part 4 will answer these questions for those who are interested in embedding Tuscany in their environment or extending Tuscany to support additional implementation and binding technologies.

Chapter 13, "Tuscany runtime architecture," will describe key aspects of the Tuscany internals to help you understand the overall architecture of Tuscany. The chapter will walk you though the primary building blocks of Tuscany and explain how they collaborate to load and run composite applications. Details about the extensibility and pluggability of Tuscany are also illustrated so you can become more familiar with how the Tuscany core interacts with Tuscany extensions. With this knowledge, you'll be able to tailor and/or extend Tuscany to meet your requirements.

If you need to add new implementation or binding types that aren't supported by Tuscany out of the box, you'll get all the help you need from chapter 14, "Extending Tuscany." This chapter covers the design and implementation of SCA implementation and binding types in detail. It gives step-by-step instructions using code snippets from existing sample binding and implementation extensions in Tuscany. For each step, we'll explain the roles played between your extension and the Tuscany core runtime and provide guidance for the design of your extension.

13 Tuscany runtime architecture

This chapter covers

- The architectural building blocks that make up the Tuscany runtime
- The complete flow of how Tuscany processes an SCA application
- The key patterns that make the Tuscany runtime flexible and extensible

Throughout this book we've promised that Tuscany handles the complexity associated with integrating various technologies in a distributed environment. You focus on developing your business logic and Tuscany handles the rest. This chapter explains the details of Tuscany's flexible architecture and reveals how the complexity is handled.

This sounds like a chapter that should appear at the start of the book. But if you're a user of Tuscany and SCA and you just want to build SCA applications, you don't necessarily need to know how the Tuscany runtime itself is architected. It's useful to know, though, if you intend to embed or extend Tuscany for use in your environment. The content of this chapter gives you a high-level understanding of

the architecture of the Tuscany runtime but with sufficient enough detail so that in the next chapter we can explain how to build Tuscany extensions.

To begin, we'll talk about the architectural building blocks that make up Tuscany and that allow you to tailor the lightweight runtime to meet your application-processing requirements. We'll then use the TuscanySCATours Payment composite from the payment-java contribution to demonstrate the various phases of application processing. This includes the initialization of the Tuscany runtime, followed by starting and managing the Payment application, and finally the shutdown process. As we go through these various phases, we'll follow the same pattern. We'll first provide a high-level view of the processing steps and then delve into more details.

After reading this chapter, you'll have a good understanding of the Tuscany architecture and will be equipped with the knowledge needed to either extend Tuscany with new functionality or to extend your own software with Tuscany's capability to develop and manage SCA applications.

13.1 An overview of the Tuscany architecture

Tuscany provides the runtime environment for deploying, running, and managing SCA composite applications. This includes the infrastructure required for handling different flavors of communication protocols used between services in a composite application. The infrastructure knows how to navigate the application and execute the services regardless of their implementation technology. Figure 13.1 illustrates the payment part of the TuscanySCATours composite application in box A. It's deployed, executed, and managed by the Tuscany runtime shown in box B.

Tuscany is a container for the SCA programming model. It's architected in a modular and extensible way. It provides core functions that handle SCA composite

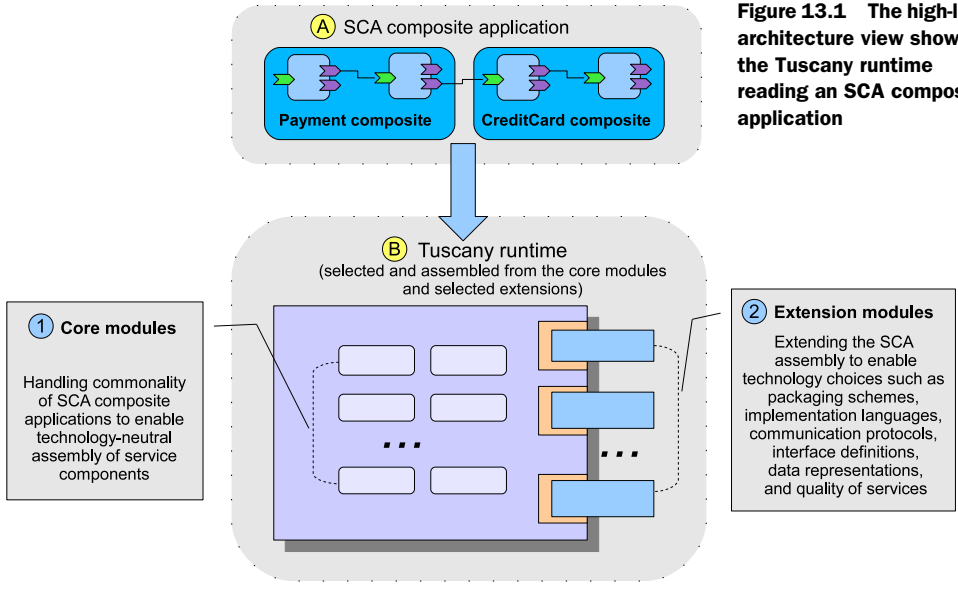


Figure 13.1 The high-level architecture view showing the Tuscany runtime reading an SCA composite application

applications and provides a mechanism for plugging in extensions to support different integration technologies such as implementation and binding types.

The Tuscany code base consists of a set of modules that can be divided into two categories: core and extension modules. These are described as boxes 1 and 2 in figure 13.1.

Think of the core modules as the foundation on which all extensions are based. The extensions are added to the core foundation to add support for various binding types, implementation types, policies, and databindings. Each extension type provides a library of choices for handling various technologies. For example, for binding types, Tuscany provides Web Services, RMI, JSON-RPC, and many other binding extensions.

Now think about the application that you're developing. Your application may not, and in most cases won't, need all the extensions. Therefore, to keep the footprint small, Tuscany allows you to choose only the extensions that you need for your composite application. As an example, if you have a component that's implemented with BPEL, you'll not need to drag in the supporting code for other implementation types. Or, if you're using only the JMS binding, there's no need to drag in Web 2.0–related bindings. You get the picture.

You'll have the same flexibility in choosing what you need when you're embedding Tuscany. You can choose to use all of the Tuscany modules or choose only the ones you need. For example, you may want to use the Tuscany modules that handle composite application assembly, but you'll need to use your own extension to handle deployment because your hosting environment uses a specific deployment model. No problem—Tuscany is architected in such a way that enables you to do this. Out of the box Tuscany already supports many different platforms, such as Apache Tomcat, Jetty, IBM WebSphere, JBoss, and Geronimo, but you can also extend Tuscany to add support for new platforms.

You now have a fairly good picture of the kind of flexibility offered by the Tuscany architecture. In the following sections we'll look at the Tuscany architecture from both structural and behavioral perspectives. The structural perspective describes the primary building blocks in the Tuscany runtime and how they're glued together to achieve modularity, extensibility, and pluggability. It gives a static view of how Tuscany is organized and assembled. The behavioral perspective describes how an SCA composite application is processed by the Tuscany runtime. Let's start with the structural perspective.

13.2 A structural perspective of the Tuscany architecture

The Tuscany code is organized as a set of modules. From a developer's point of view, a module is a project in an Eclipse IDE, or an artifact in a Maven build, that's used to produce a JAR file. It usually takes more than one of the JARs in the Tuscany distribution to support a single function in the Tuscany runtime. As an example, look at figure 13.2. The Web Services extension shown in box 2 consists of three JAR files: binding-ws, binding-ws-xml, and binding-ws-axis2. All three are needed when using `binding.ws` in your composite applications. The binding-ws-axis2 module can be replaced with another Web Services provider such as Apache CXF without affecting how the user

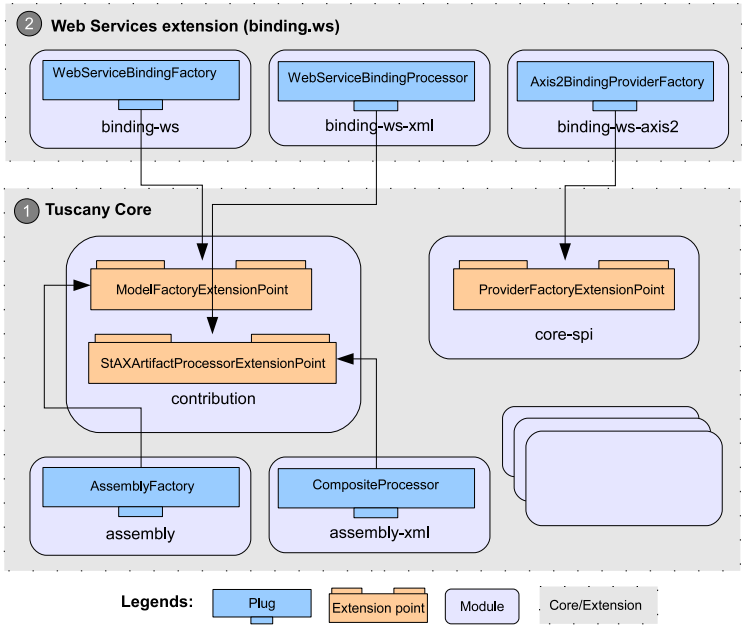


Figure 13.2 Plugging the Web Services binding and Axis2 extension into the Tuscany runtime extension points

exploits the SCA `<binding.ws>` element. We don't have a binding-ws-cxf yet, but if you decided to write some CXF integration code, this is where you'd put it.

We mentioned earlier that the core modules, some of which are shown in box 1, provide a framework for extending the SCA programming model and adding extensions. The framework defines a set of common interfaces and resources. For each extension, the extension modules implement these common interfaces and provide the extension-specific resources. The places in the Tuscany framework where supporting code or resources are added are called extension points. We'll call the extension code itself *plug-in* code to distinguish it from our use of the word *extension* to refer to implementation-agnostic extension elements, such as `<binding.ws>`.

Let's look at figure 13.2 again and try to identify the extension points and the plug-ins. In the Tuscany core, in box 1, the Tuscany runtime defines two contribution extension points where extensions provide the plug-in code that supports the loading of the extension as it appears in the composite file. In this case, two plug-ins are provided to support loading of `<binding.ws>` elements. The extension point in the core-spi module allows the runtime to manage the lifecycle of the Web Services binding.

Let's take a closer look at what constitutes the Tuscany core and Tuscany extensions.

13.2.1 Tuscany core functions

The Tuscany core is responsible for bootstrapping the runtime and for loading, running, and managing SCA composite applications. During the bootstrap phase, the Tuscany core discovers the available extensions and uses this information to load the composite application. The Tuscany core builds an in-memory representation of the composite application based on what it reads from the composite file. As the core

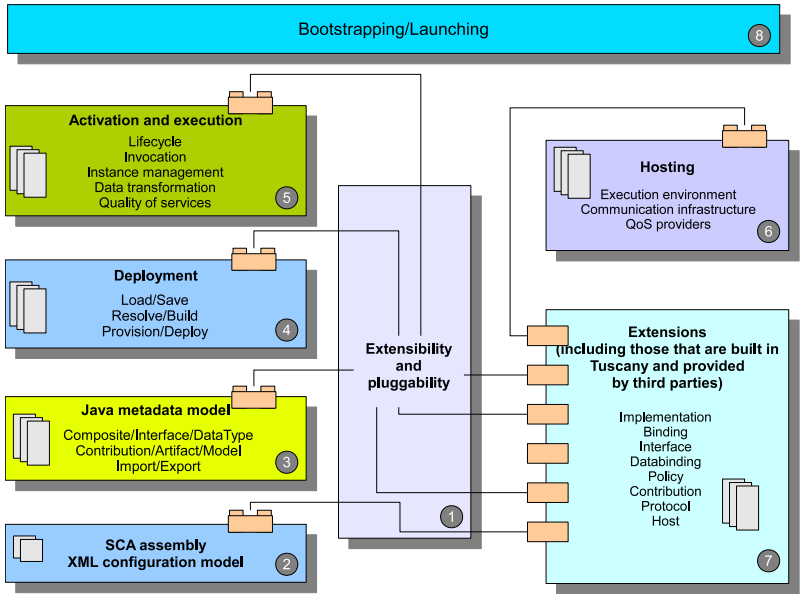


Figure 13.3 Key functional blocks in the Tuscany runtime

reads the composite file, it identifies which extensions it needs in order to run the application. It then delegates the work to the appropriate extensions when the application is run. For example, if the composite application uses a BPEL process to implement a component, the core delegates the work of loading and executing the process to the ODE BPEL engine. The building blocks that make up the Tuscany core are illustrated in figure 13.3.

The code in box 1 enables extension points and plug-ins to be declared and discovered. The content of the other boxes defines the extension points themselves and the contract between the extension points and the plug-ins.

Box 2 is the XML configuration model for an SCA composite application. The default configuration model is defined by the SCA Assembly Model Specification as a set of XML schemas and can be extended to add new implementation types, binding types, and so on.

Box 3 consists of a set of interfaces and default implementations for interfaces that describe the in-memory model of an SCA composite application. For example, the Java models for Composite, Component, Implementation, Service, Reference, and Binding are in the SCA assembly module under box 3. Tuscany uses these models to represent the components of a composite application.

Box 4 consists of a set of modules that handle deployment of SCA composite applications within the SCA domain. An SCA composite application is packaged as one or more archives called SCA contributions. Tuscany provides the code to scan SCA contributions and load artifacts such as composite files, Java classes, and WSDL/XSD documents into memory using the metadata models. It resolves references in the model across artifacts and contributions following import and export directives. After the resolution, we'll have an in-memory graph of the composite application.

Box 5 is the supporting code for the core runtime, which handles lifecycle, invocation, instance-management, and data-transformation capabilities for SCA components. It also includes some SPIs for plugging in extensions. The core runtime collaborates with extensions to start and stop SCA services, references, and implementation instances and to drive the invocations from and to SCA components.

Box 7 represents the different extensions that can be plugged into the Tuscany runtime to support different kinds of implementations, bindings, protocol, hosts, interfaces, policies, databindings, and packaging schemes.

The Tuscany runtime can be used in different hosting environments. The supporting code for this resides in box 6. Out of the box, Tuscany provides integration with various environments such as Java SE, Tomcat, Jetty, OSGi, and Java EE so that it can be either embedded within application servers or used to deploy and run SCA composite applications in a standalone mode. This can also be extended to support new platforms.

Box 8 contains the set of Tuscany APIs and tools that the end user uses to bootstrap Tuscany and launch SCA applications.

In the next section, we'll concentrate on box 1 and explain how Tuscany supports the plugging in of extensions.

13.2.2 Tuscany runtime extension points and plug-ins

As you already know, the SCA programming model allows different technologies to be used in an SCA assembly through its extensible programming model. It defines extension types such as implementation, binding, interfaces, and policies. This creates an interesting challenge for Tuscany. It must take care of the anomalies of each technology while going through the various phases of invocation, execution, and management of a composite application.

Figure 13.4 shows an example of how the Tuscany runtime handles various packaging schemes for an SCA contribution, such as a filesystem directory, an OSGi bundle, a compressed directory (zip, gzip, and so on) or a JAR file (or its variants—WAR, EAR, and the like).

Tuscany meets the extensibility challenge by defining a consistent and simple model for adding extensions through its extension point mechanism. An extension point is a bridge between the core and extensions that are not known to the core upfront. It functions as a container for a given extension type and collects all the extensions of that type that plugs into the Tuscany runtime. An extension point usually comes with a Java interface or base class to define the behavior of the extensions.

In figure 13.4 Tuscany defines a generic `ContributionScanner` interface that's implemented for each packaging scheme. Each `ContributionScanner` scans the specific packaging format to build a list of artifacts. Each new `ContributionScanner` implementation is registered with the Tuscany runtime via the `ContributionScannerExtensionPoint` extension point. This is a known location where the Tuscany runtime looks for information about supported contribution packaging formats. During contribution processing, the Tuscany runtime uses the `ContributionScannerExtensionPoint` to find the contribution scanner that knows how to handle the

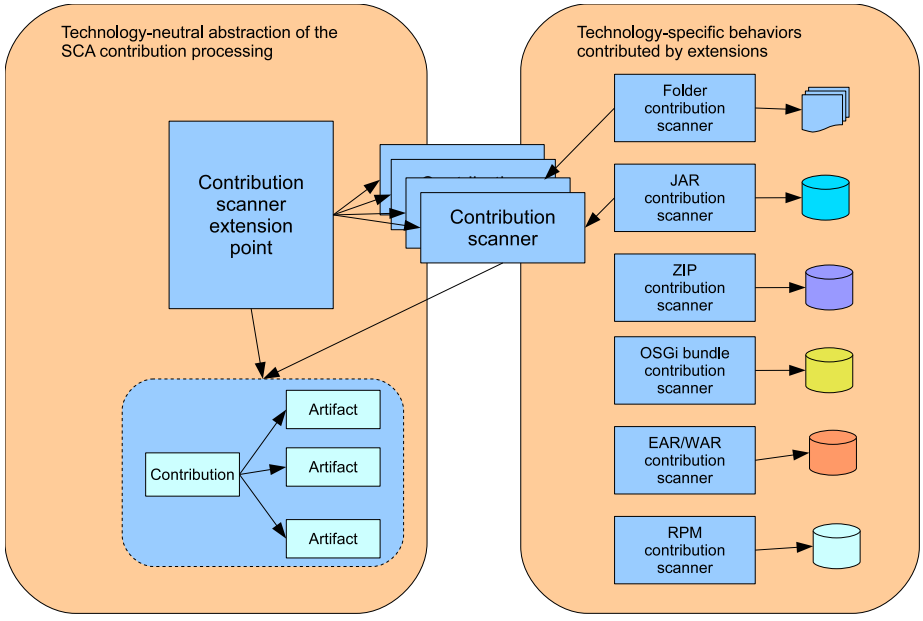


Figure 13.4 The contribution scanner extension point and contribution scanner extensions

required packaging format. The runtime delegates handling of the contribution to the scanner.

At the heart of the Tuscany runtime is an extension point registry that holds a list of all the extension points. This is shown on the left side of figure 13.5.

The `StAXArtifactProcessorExtensionPoint` is used to register the XML and Java configuration model for each extension. As you can see in figure 13.5, the Java implementation extension and Web Services binding extension both register plug-in code with `StaxArtifactProcesssorExtensionPoint`.

The extension point registry will be used to find information about extension points and find extension plug-in code. The following piece of code shows how the registry lookup works.

```
ModelFactoryExtensionPoint factories =
    registry.getExtensionPoint(ModelFactoryExtensionPoint.class);

assemblyFactory = new RuntimeAssemblyFactory();
factories.addFactory(assemblyFactory);

MessageFactory messageFactory =
    factories.getFactory(MessageFactory.class);
```

First, the extension point is looked up in the registry using the extension point's interface, in this case `ModelFactoryExtensionPoint`. Then a new plug-in is added into this factory extension point. The plug-in here is `RuntimeAssemblyFactory`. Finally the extension point is used to look up a factory plug-in based on the type of thing that the factory can create, in this case `MessageFactory`-based classes.

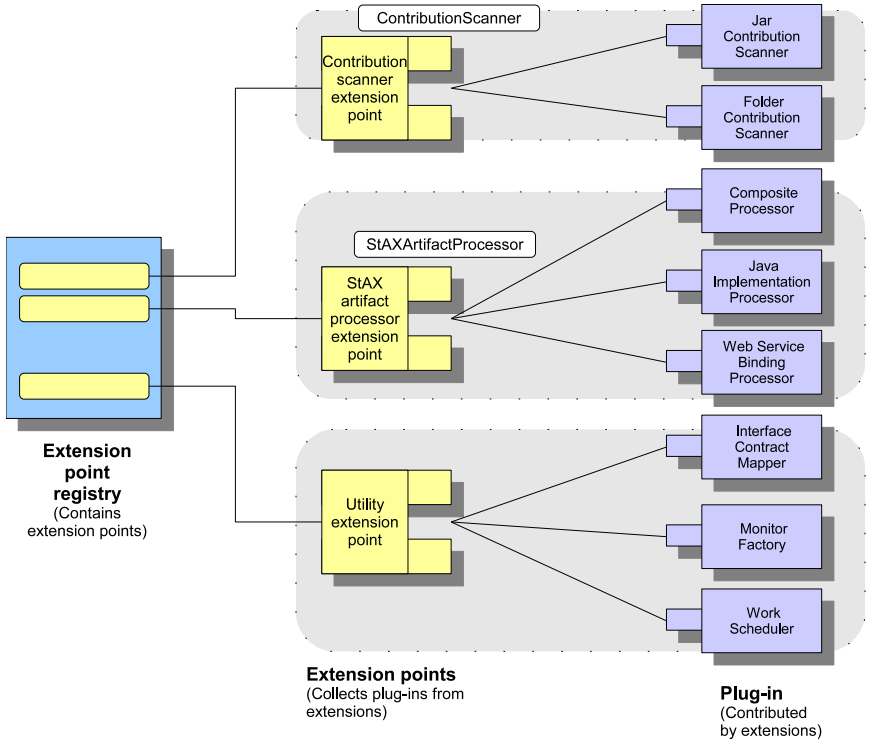


Figure 13.5 The ExtensionPointRegistry contains pointers to extension points that support the various functional requirements of the extensions.

Now that you understand the idea of extension points and plug-in code, we'll look at how the Tuscany runtime loads the extension points and the plug-in code associated with them and utilizes the information to process composite applications.

13.2.3 Defining extension points and plug-ins

In Tuscany, extension points and plug-in code are modeled using Java interfaces or abstract classes. To allow extension points and plug-in code to be declarative and pluggable, Tuscany adopts the service provider pattern defined by the JAR file specification.

> **JAR file service provider**
>
> The JAR file specification defines a mechanism to declare service providers using configuration files under the META-INF/services directory. For more details, please see the following site:
>
> http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html#Service Provider

In the context of this pattern, implementations of extension points and plug-ins are service providers. They identify themselves to the Tuscany runtime by providing a

configuration file that resides in the resource directory META-INF/services. The name of the configuration file should be the name of the fully qualified Java class name representing the extension point or plug-in. The file should contain a newline-separated list of unique and concrete provider-class names and must be encoded in UTF-8. Each line declares a provider for the named extension point.

An example is shown in figure 13.6. The AssemblyFactory is the extension point responsible for creating Java models for the SCA assembly entities such as Composite or Component. `AssemblyFactory` is the service provider interface and `DefaultAssemblyFactory` is the service provider that contains the concrete implementation for the `AssemblyFactory` interface.

The `DefaultAssemblyFactory` class is declared to Tuscany using a plain-text file named META-INF/services/org.apache.tuscany.sca.assembly.AssemblyFactory. The content of the file is as follows:

```
org.apache.tuscany.sca.assembly.DefaultAssemblyFactory
```

Some plug-ins can be declared with additional attributes so that multiple instances can be identified by the owning extension point. For example, the `CompositeProcessor` can be registered in the `StAXArtifactProcessorExtensionPoint` using a services file called META-INF/services/org.apache.tuscany.sca.contribution.processor.StAXArtifactProcessor:

```
org.apache.tuscany.sca.assembly.xml.CompositeProcessor;qname=
➥ http://www.osoa.org/xmlns/sca/1.0#composite,model=
➥ org.apache.tuscany.sca.assembly.Composite
```

The `qname` attribute allows the `StAXArtifactProcessor` extension point to find a `CompositeProcessor` using the XML element name of the extension being processed.

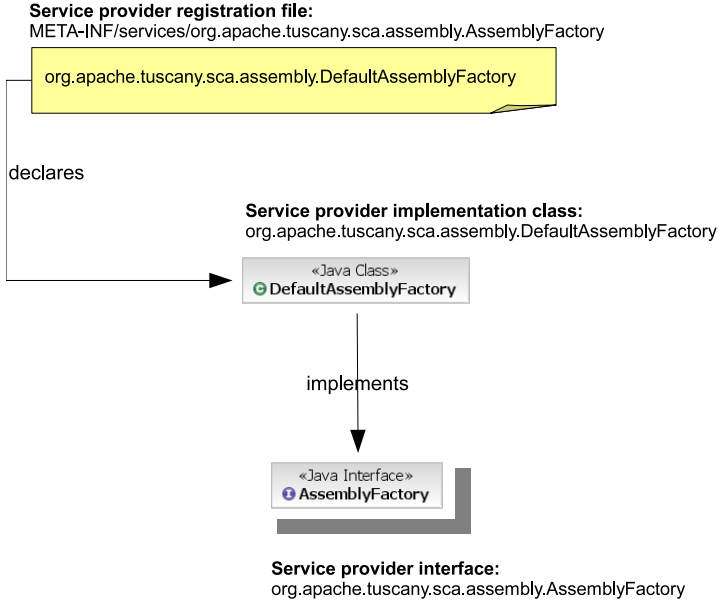


Figure 13.6 The service provider interface, implementation, and declaration

The model attribute associates the StAX processor, `CompositeProcessor` in this case, with the Java model interface it can handle.

When the runtime realizes the need for a given extension, it looks it up by using the Tuscany `ServiceDiscovery` SPI. The corresponding code is shown in the next snippet:

```
ServiceDeclaration factoryDeclaration =
ServiceDiscovery.getInstance().
getFirstServiceDeclaration(factoryInterface.getName());
if (factoryDeclaration != null) {
Class<?> factoryClass = factoryDeclaration.loadClass();
...
}
```

Figure 13.7 shows how the Tuscany runtime discovers plug-ins. Tuscany calls the Tuscany `ServiceDiscovery` SPI to discover, load, and instantiate the implementation class for a given plug-in interface.

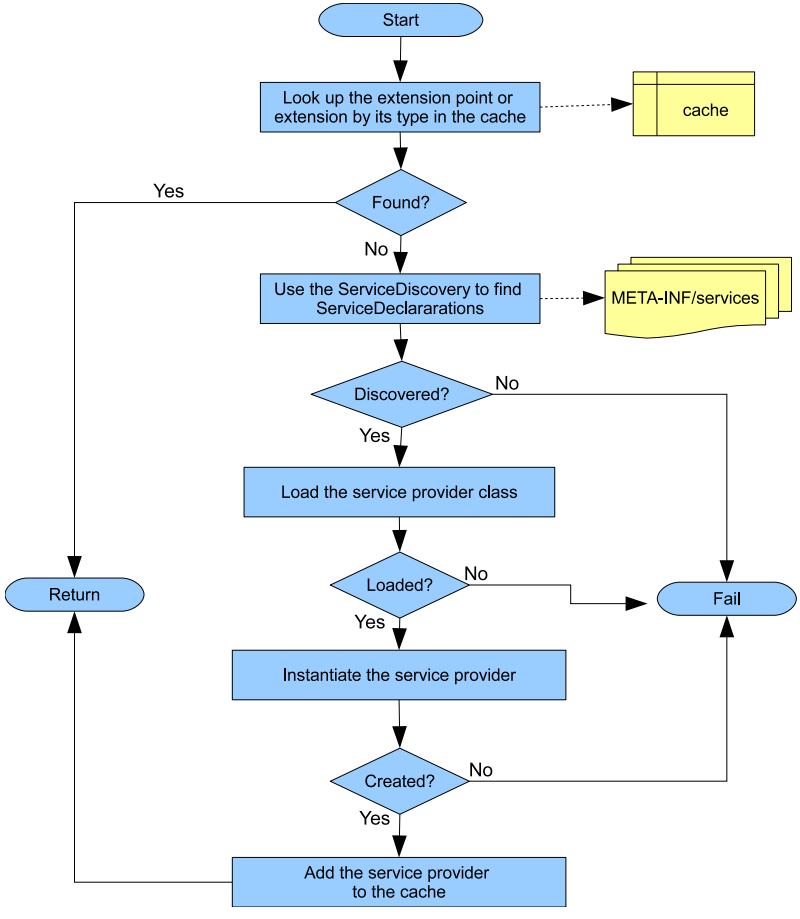


Figure 13.7 The Tuscany runtime's service discovery and plug-in instantiation flow. In this flow the service provider refers to the plug-in code providing a service in the context of the JAR file service provider pattern.

You now understand how the Tuscany runtime provides a modular, extensible, and pluggable infrastructure. Next, let's focus on how the various functional pieces collaborate with one another to process an SCA composite application. We'll walk you through the steps that the Tuscany runtime takes to process an SCA composite application. You'll learn how the building blocks we've discussed in this subsection work with each other.

13.3 A behavioral perspective of the Tuscany architecture

As a user of the Tuscany runtime, you'll configure an execution node with a list of contributions, either on the command line or via the domain manager. Once the node is started, all you'll see is that component services are made available. Under the covers, a sequence of steps is performed in order to load the provided contributions, read any composites that they contain, and process the components of the composites in order to create and expose the component services.

The steps are shown in figure 13.8. The object of these steps is to create an in-memory model of the SCA components described in the contribution's deployable composite files. The components in this model communicate with one another and collaborate to perform the task that the composite application is designated to handle.

As shown in figure 13.8, the first step of processing bootstraps the Tuscany kernel in the host environment. In this step, Tuscany creates an extension point registry, which will be used to discover and load extension points and plug-ins on demand. At this point, the Tuscany runtime is ready to process composite applications.

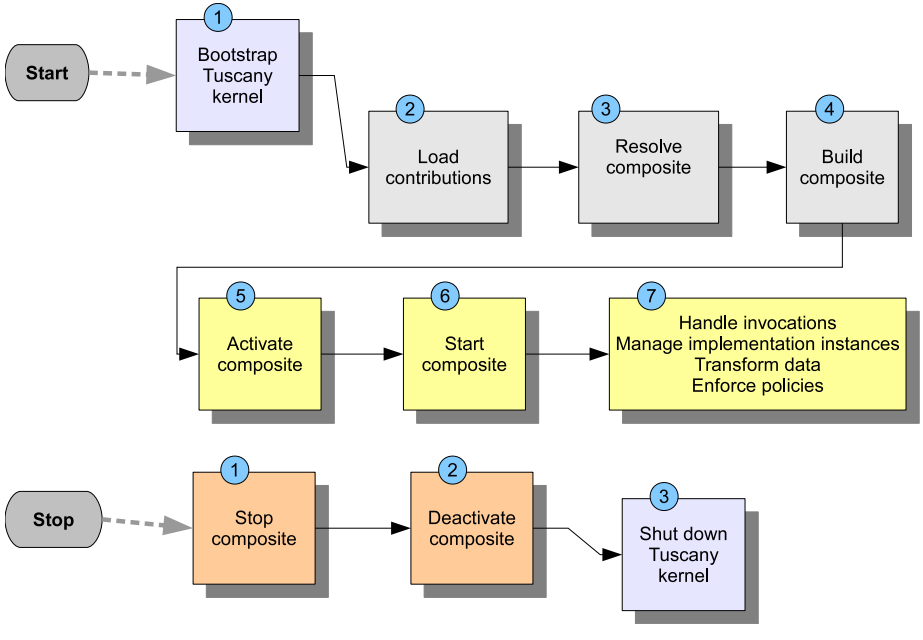


Figure 13.8 The Tuscany runtime steps for starting and stopping an SCA composite application

In the second step, Tuscany loads the composite application using a list of SCA contributions. The Tuscany runtime uses `ContributionScanner` plug-ins to build a list of artifacts from the contributions. Examples of artifacts include SCA composite files, WSDL/XSD files, Java classes, and BPEL processes. These artifacts are represented in memory as a series of Java models. The model of the composite files and the components they contain is referred to as the assembly model because it models the SCA assembly.

Tuscany has now populated the artifact models. A model, such as an SCA composite file, can reference other models by name or location. In the third step, the Tuscany runtime finds such by-name references and replaces them with references to concrete model objects. This step is called resolution. For example, a Java class name gets linked to its implementation class, or the XML-qualified name of an included composite is replaced with the real composite model. All references need to be mapped to real objects before the Tuscany runtime proceeds to the next step.

The SCA programming model is quite flexible, and it enables components to inherit settings or to override them. In step 4, the Tuscany runtime refines, or builds, the assembly model and applies the inheritance and scope rules. At the end of this step, you'll have a complete view of the SCA composition, including service endpoints, references to service endpoints, service implementations, nested composite resolution, and policy settings.

In the fifth step, the Tuscany runtime activates the SCA composites by creating runtime provider artifacts using the provider plug-in code from implementation, binding, and policy extensions. Providers handle lifecycle events and add extra functions to the invocation path on behalf of the extensions. As a result of this step, the invocation path between each SCA reference and service is set up.

In the sixth step, the Tuscany runtime starts the SCA composites by delegating the start event to the provider code associated with component implementations, reference bindings, service bindings, and policies. By the end of this step, SCA components in the composite are ready to accept incoming requests, call the implementation instances, and invoke other services.

The seventh step handles the interaction between components. The inbound path starts when an SCA component is accessed from the service binding, and the outbound path starts when an SCA component invokes another service via the reference binding. In step 7, a chain of runtime message handlers is utilized to transform data, process business logic, enforce policies, and dispatch outbound calls to the protocol stack.

The stop flow shown in figure 13.8 is simpler. Tuscany first asks the providers of SCA components to stop SCA services, references, and implementations and release any resources associated with these objects. The second step disassociates the provider code from the assembly model. The third step shuts down the Tuscany kernel. All the modules are stopped. Extension points and plug-ins have a chance to perform cleanup here.

Now you have a high-level view of how Tuscany processes composite applications. We'll look in more detail at the key steps in the processing.

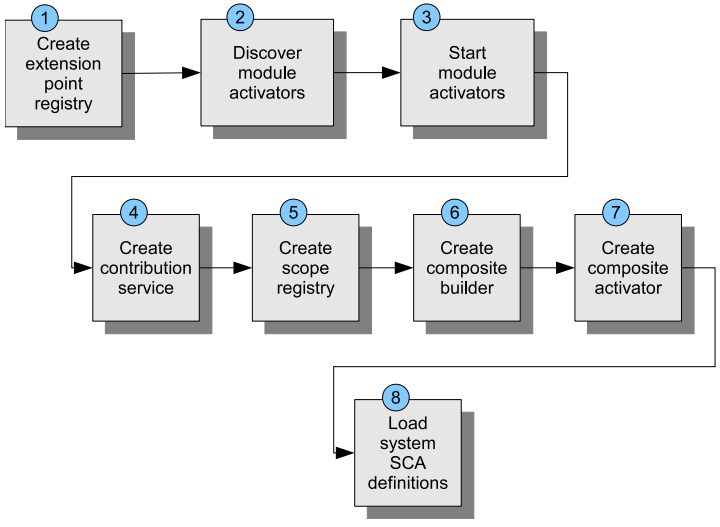


Figure 13.9 Tuscany's bootstrap sequence

13.3.1 Starting and stopping the Tuscany runtime

A Tuscany node is the runtime environment in which the SCA composite application is run. Before the node loads the application, it needs to create the extension point registry so that extension points and plug-in code can be discovered, loaded, and called. This is step 1, as shown in figure 13.9.

The second step discovers the list of module activators that help Tuscany to handle the lifecycle of core and extension code. Both core and extensions can have one or more module activators that are implementations of the following interface:

```
public interface ModuleActivator {
    void start(ExtensionPointRegistry registry);
    void stop(ExtensionPointRegistry registry);
}
```

You can register the plug-in code for an extension's module activator by adding a line to the configuration file named META-INF/services/org.apache.tuscany.sca.core.ModuleActivator and providing the full path of the implementation class that handles activation. For example, the following line declares a module activator for `implementation.java`:

```
org.apache.tuscany.sca.implementation.java.JavaImplementationActivator
```

The Tuscany runtime runs the module activators in step 3. In steps 4–7, it creates utilities to load contributions, build composites, and activate and start composites. Tuscany loads the system-level SCA policy definitions in step 8. After that, Tuscany is ready to load an SCA application.

The Tuscany runtime can be shut down once the application is stopped and deactivated. This step releases resources that are allocated by the extension points and plug-in code and calls the `stop()` method of all the module activators.

13.3.2 Loading SCA applications

The Tuscany runtime reads contributions to form an in-memory representation of the SCA composite application. We'll again use the payment composite application to help you understand the process. The payment application is an SCA contribution packaged as a JAR file. The contribution contains Java classes, composite files, and other artifacts such as WSDL documents. The Tuscany runtime goes about processing the application in the order discussed in the next section.

SCA CONTRIBUTION PROCESSING OVERVIEW

Figure 13.10 illustrates how the Tuscany runtime processes the payment contribution.

In step 1, the payment.jar file is scanned to build a list of artifacts, including the composite file. Step 2 finds artifacts of interest and loads them into memory as Java models. As shown in step 3, for composite files like payment.composite, the Tuscany runtime delegates the loading of the XML elements, such as `<composite>` or `<implementation.java>`, to a set of XML processors. Tuscany also introspects component implementations and interfaces to discover more metadata. After all the contribution artifacts have been read, references between them can be resolved. Let's look more closely these steps.

SCANNING AN SCA CONTRIBUTION

In figure 13.10, the left side shows the `ContributionScanner` interface. This is plug-in code that's implemented to handle specific packaging formats, such as filesystem directory, JAR, and zip. Scanning a contribution produces a list of artifacts such as Java classes, SCA composite files, WSDLs, and XSDs. Each implementation of the

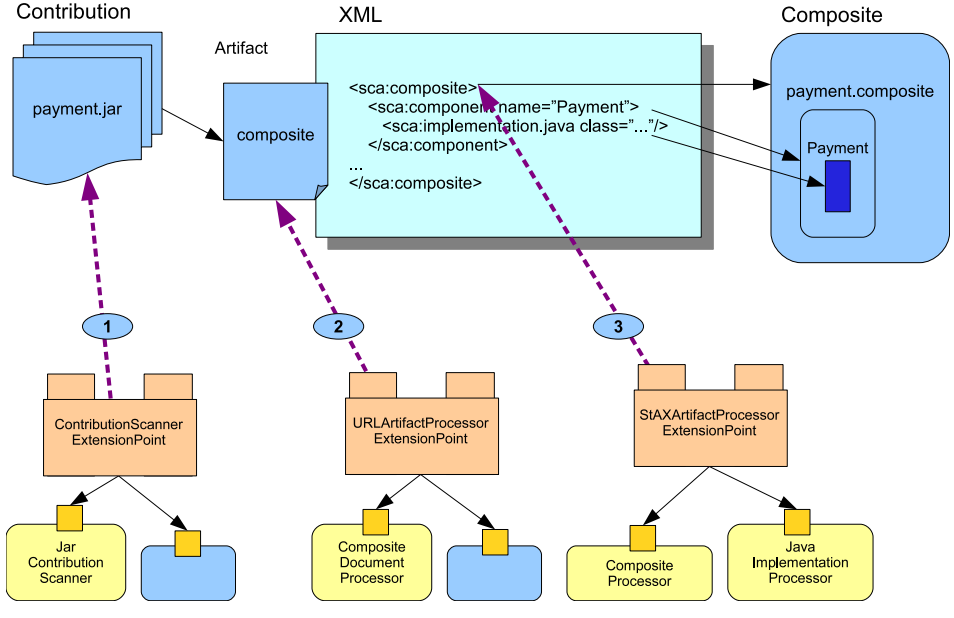


Figure 13.10 Tuscany's bootstrap sequence

`ContributionScanner` interface is registered with the `ContributionScanner-ExtensionPoint`. This was described in detail in section 13.2.2.

LOADING ARTIFACTS

The Tuscany runtime uses `URLArtifactProcessorExtensionPoint` plug-ins to parse the artifacts found in the contribution. As shown in figure 13.10, the `Composite-DocumentProcessor` is registered with the runtime and knows how to parse composite documents such as the payment.composite file.

PROCESSING XML ARTIFACTS USING STAX

If the artifact is an XML document, such as an SCA composite file, the `URLArtifact-Processor` will locate a relevant `StAXArtifactProcessor` using the QName of each XML element. The parsed information is used to create a Java model. In our payment example, distinct processors are used to process the `<composite>` and `<implementation.java>` elements. The interactions between various extension points and plug-ins are illustrated in figure 13.11.

The `<composite>` and `<component>` elements are handled by the `CompositePro-cessor` and `AssemblyFactory`, whereas the `<implementation.java>` element is processed by the `JavaImplementationProcessor` and `JavaImplementationFactory`.

INTROSPECTING JAVA CLASSES AND INTERFACES

Java interfaces and implementation classes can contain Java annotations that relay information about the assembly of the application, such as remotability of a Java interface,

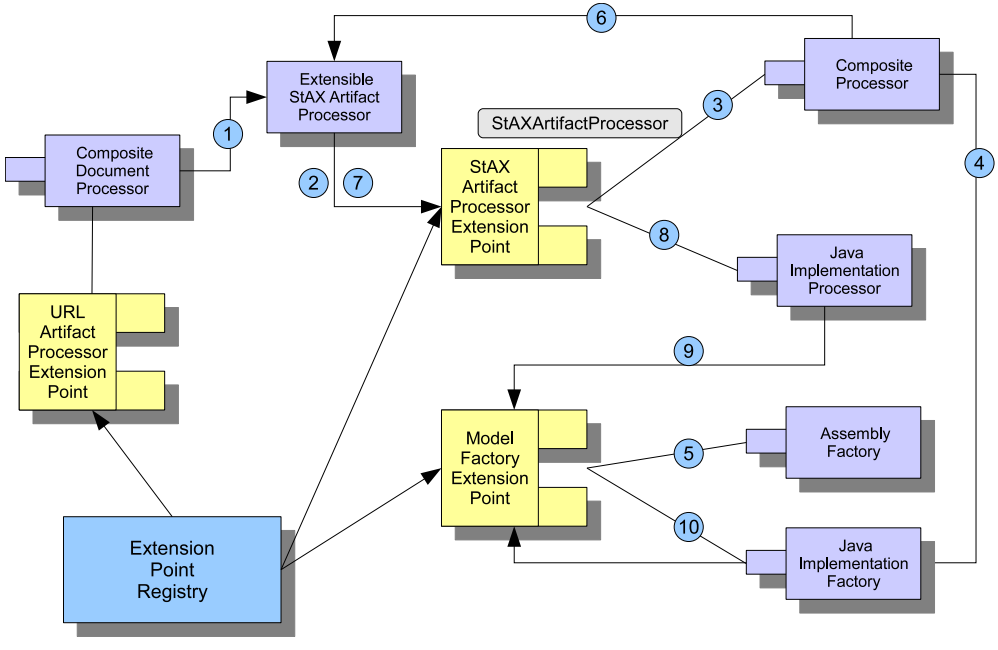


Figure 13.11 Collaboration between extension points and plug-ins. The arrows with numbers show the sequence of lookups and delegations.

intents, policy sets, properties, reference, services, and WSDL/XSD mappings. Tuscany defines an extension point that collects plug-ins implementing the `JavaInterfaceVisitor` interface. For an SCA component implemented in Java, the implementation class is introspected using a set of annotation processors that implement the `JavaClassVisitor` interface.

RESOLVING REFERENCES IN JAVA MODELS

The SCA assembly model is a graph of related artifacts that can reference each other by name, location, and other symbolic links. Tuscany uses the proxy pattern to seamlessly connect the model objects. Figure 13.12 shows an example where the CreditCardPayment service is configured with a WSDL interface by the QName of the WSDL portType.

Let's look at how Tuscany handles such object references. Creditcard.composite and CreditCardPayment.wsdl are two files in the contribution. In step 1 of figure 13.12, the composite file is loaded into memory, and a `WSDLInterface` object (A) is created to hold the QName of the WSDL portType. Please note that the portType itself isn't set. Step 2 loads the WSDL file and creates a WSDL `portType` object (B), which contains the definitions. In step 2, the WSDL `portType` object is added to the WSDLModelResolver, which is the plug-in code that knows how to find WSDL elements by QName.

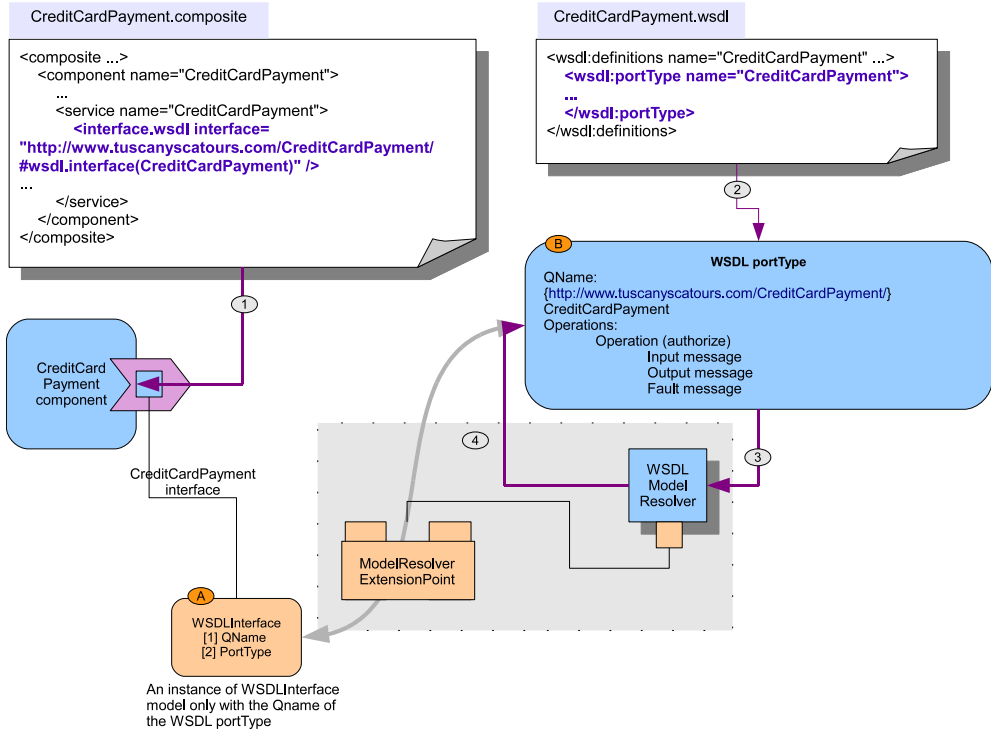


Figure 13.12 Tuscany uses the ModelResolverExtensionPoint and specific ModelResolver plug-in code to connect the WSDLInterface to a concrete WSDL `portType` object by QName.

When Tuscany asks the StAX processor to resolve the CreditCardPayment component's references to other objects, the processor looks up the `WSDLModelResolver` from the `ModelResolverExtensionPoint`. The `WSDLModelResolver` returns the WSDL portType, matching the QName provided, to the processor, and the processor sets it onto the portType field of the `WSDLInterface` object. Now the `WSDLInterface` is resolved, and it contains the complete information of the WSDL portType.

After the resolution, you'll have a complete in-memory representation of the SCA composites and all their referenced artifacts such as Java interfaces and classes, WSDLs, XSDs, BPEL processes, binding configurations, intents, and policy sets.

13.3.3 Building SCA composites

In the hierarchy of an SCA composition, some configuration is inherited. Figure 13.13 shows two types of inheritance: structural inheritance and implementation inheritance.

Implementation inheritance occurs when a component receives configuration from its component type. For example, when using `implementation.composite`, the component type is built through the promotion of services and references from the named composite.

Structural inheritance describes the relationship between elements of the XML tree in the composite file. In the structural inheritance hierarchy, configuration at a higher-level element of the XML tree is applicable to its descendant elements. For example, a required intent at a composite element applies to the component elements and their descendants.

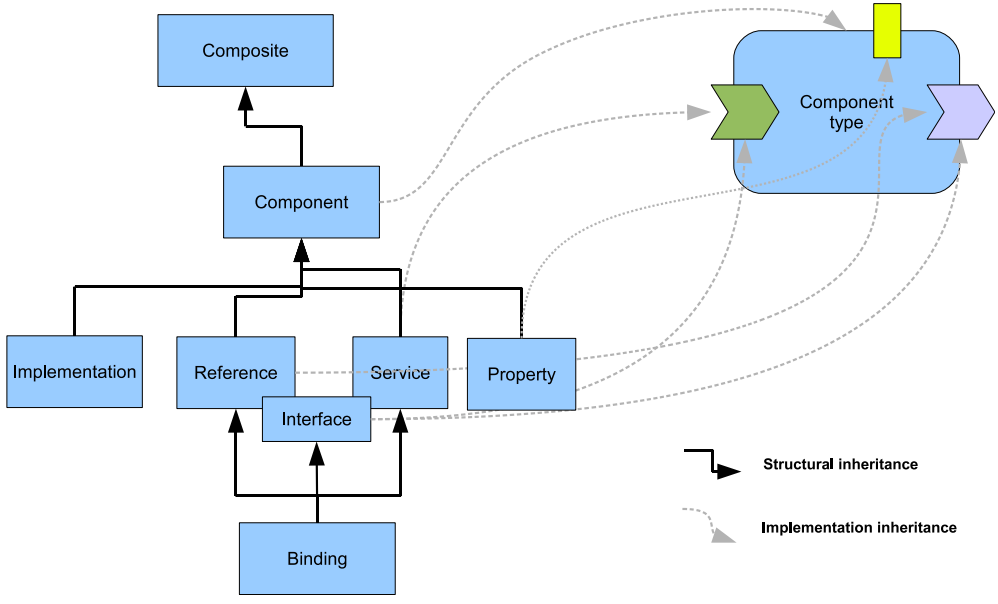


Figure 13.13 Configuration inheritances following the structural and implementation hierarchies

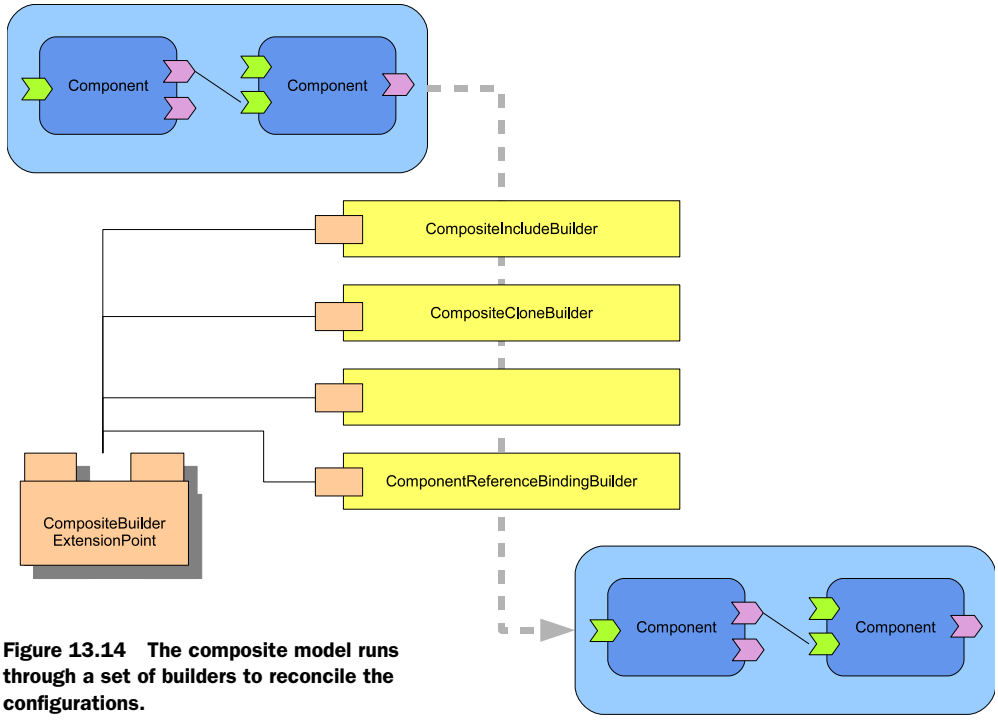


Figure 13.14 The composite model runs through a set of builders to reconcile the configurations.

Tuscany uses builders to walk through the composite model and apply inheritance rules to and finalize the true shape of the composite application. This is illustrated in figure 13.14.

The builders, such as `CompositeIncludeBuilder` and `CompositeCloneBuilder`, collaborate with one another to determine the final configuration of the composite application with all inheritance rules applied. The output of each builder is fed into the next one in the chain.

At the end of build phase, the final configuration is available for component implementations, service bindings, and reference bindings. Tuscany will use this information to drive component lifecycles, invocations, and policy enforcements.

13.3.4 Augmenting the composite with runtime artifacts

An SCA component's implementation instance can invoke another service via a reference binding or respond to requests via a service binding. Such invocations can be intercepted to perform tasks such as data transformation, user authentication and authorization, or transaction demarcation. The interceptors used to perform these functions are typically contributed to the runtime as a set of providers by various extensions. The Tuscany core associates these providers with the SCA in-memory assembly model and sets up the chains of interceptors that are called during the invocation process. Figure 13.15 illustrates a chain of interceptors for the invocation process.

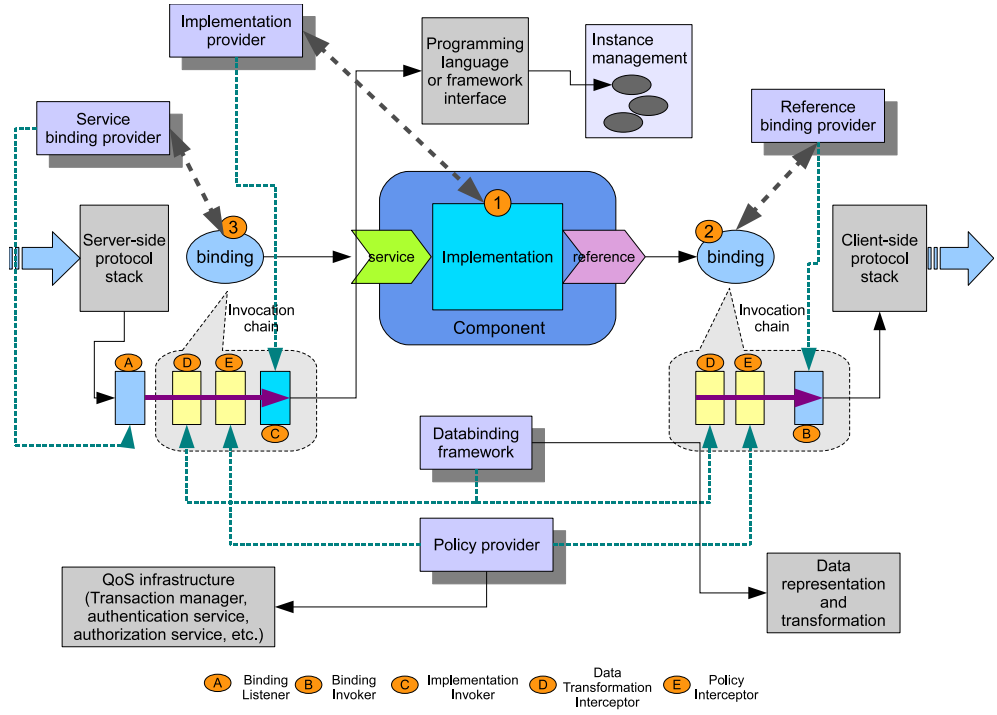


Figure 13.15 Runtime providers enable the Tuscany runtime to control the lifecycle of an SCA component, interact with communication protocols, invoke the business logic, transform data, and force quality of services. They connect the SCA programming model constructs to the plumbing technologies.

Various types of providers connect an SCA component to the underlying technologies on which it's built. They do this by responding to lifecycle events and adding functions to the invocation path:

- Implementation providers interact with the programming language or framework that's used to implement the business logic.
- Binding providers handle outbound communications from SCA references and inbound communications to SCA services over the binding protocols.
- Policy providers handle the infrastructure-specific policy associated with the quality of service handling.
- Data transformation providers handle different representations of data and transform data from one format to the other.

Let's look at how Tuscany handles component interactions using a chain of interceptors.

RUNTIME CONSTRUCTS FOR SCA COMPONENT INTERACTIONS

The Tuscany runtime adopts the chain of responsibilities pattern and introduces `InvocationChain`, `Invoker`, `Interceptor`, and `Message` concepts to allow a list of functions to be called in sequence to process a message. This is illustrated in figure 13.16.

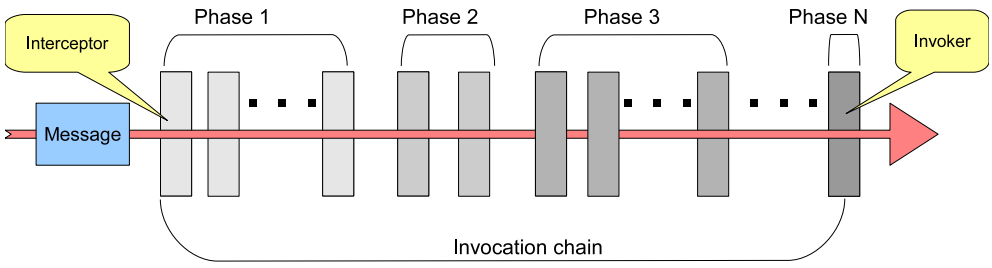


Figure 13.16 `Interceptors` and `Invokers` are ordered by phases in the invocation chain.

Tuscany defines a `Message` interface to hold the data to be exchanged. A `Message` contains a body and a list of headers. The body is used to carry the business data, whereas headers are used to pass extra information such as security subjects and endpoint descriptions. Tuscany also defines a `MessageFactory` interface to create instances of `Message`.

An `Invoker` is the basic unit that can process a request message and produce a response message. The interface is shown in the following snippet:

```
public interface Invoker {
    Message invoke(Message msg);
}
```

If a function needs to know the next `Invoker` in the pipeline, it needs to implement the `Interceptor` interface, which adds two methods to `Invoker` so that the next `Invoker` can be set and retrieved. The following snippet shows the `Interceptor` interface:

```
public interface Interceptor extends Invoker {
    void setNext(Invoker next);
    Invoker getNext();
}
```

The Tuscany runtime creates an `InvocationChain` for invoking an operation related to a reference or service binding. The `InvocationChain` links all the `Invoker`s and `Interceptor`s and exposes the head `Invoker` of the chain:

```
public interface InvocationChain {
...
    void addInterceptor(Interceptor interceptor);
    void addInvoker(Invoker invoker);
    Invoker getHeadInvoker();
    void addInterceptor(String phase, Interceptor interceptor);
}
```

Within the same invocation chain, some interceptors are required to be executed before or after other ones. For example, on the service-binding side, the data needs to be transformed by the `DataTransformationInterceptor` before control is handed to the `ImplementationInvoker` that interacts with an implementation instance. Tuscany uses the concept of a *phase* to order the invokers in the invocation chain. A phase represents a group of `Invoker`s, and each phase has a position in the chain. Tuscany has

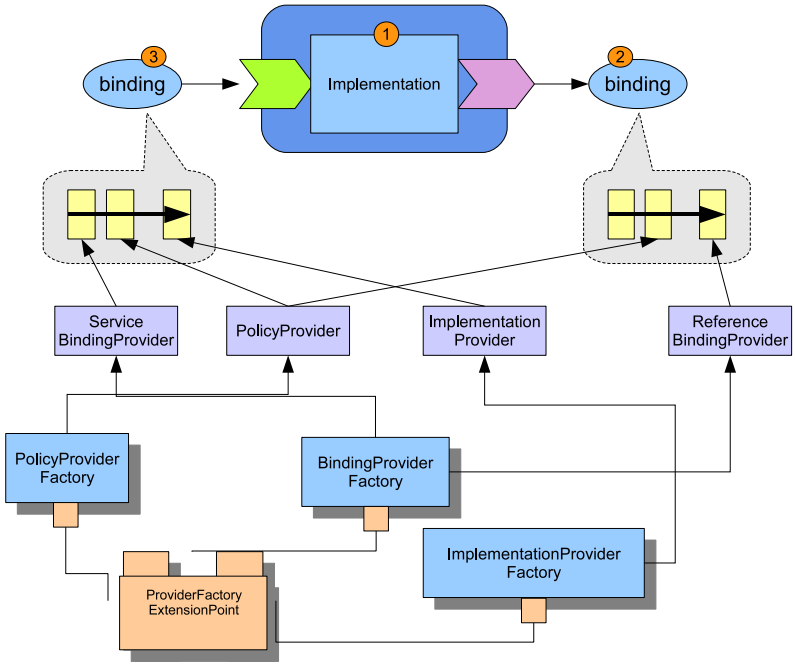


Figure 13.17 Implementation, Binding, and Policy extensions contribute `Invokers` to the invocation chains.

built-in phases that can be extended through the PhaseExtensionPoint. `Invoker`s are added by phase and are ordered by the phase. `Invoker`s within the same phase can't assume any order.

BUILDING THE INVOCATION CHAINS FOR REFERENCES AND SERVICES

The Tuscany core works with extension providers to set up the invocation chain and handle the lifecycle of SCA components. The ProviderFactoryExtensionPoint is shown in figure 13.17.

The runtime utility called `CompositeActivator` is responsible for finding the provider factories for implementation, binding, and policy types and associating them with component implementations, references, and services. `CompositeActivator` calls the implementation provider for the service side to add an `Invoker` to dispatch invocation requests to the implementation instances. It calls the reference-binding provider to contribute an `Invoker` to the reference side to handle outbound invocations through the underlying protocol stack. Other interceptors can also be inserted into the invocation chain by policy providers to add logic to enforce quality of services by intercepting the invocations.

Implementation and reference-binding providers have callback methods to handle the lifecycle events when the corresponding SCA component implementation, reference binding, and service binding are started or stopped by Tuscany runtime. Let's see what happens when an SCA component is started or stopped.

13.3.5 Starting and stopping an SCA component

An SCA component needs to be started before it can interact with other services. Starting a component will start the component reference bindings, service bindings, its implementation, and associated policies. This allows the binding, implementation, and policy providers to set up the environment to handle invocations and policies.

STARTING AND STOPPING AN SCA COMPONENT IMPLEMENTATION

Each implementation provider sets up the environment required for instantiating and invoking implementation instances. For example, the provider for `implementation.java` handles dependency injection. Figure 13.18 shows how this is done.

SCA Java components can use Java annotations to define references, properties, and context. When a Java component is started, the `JavaImplementationProvider` finds the corresponding fields, setter methods, or parameters of a constructor, creates injectors, and associates them with `ObjectFactory` instances that can be used to create values or proxies for the injection.

When a Java implementation instance is instantiated, the runtime creates a value for a Java property using the `JavaPropertyValueObjectFactory`. It creates a proxy for SCA references and callbacks using the `ProxyFactoryExtensionPoint`, which holds `JDKProxyFactory` and `CglibProxyFactory` instances. Such values or proxies are then injected into the component implementation instance.

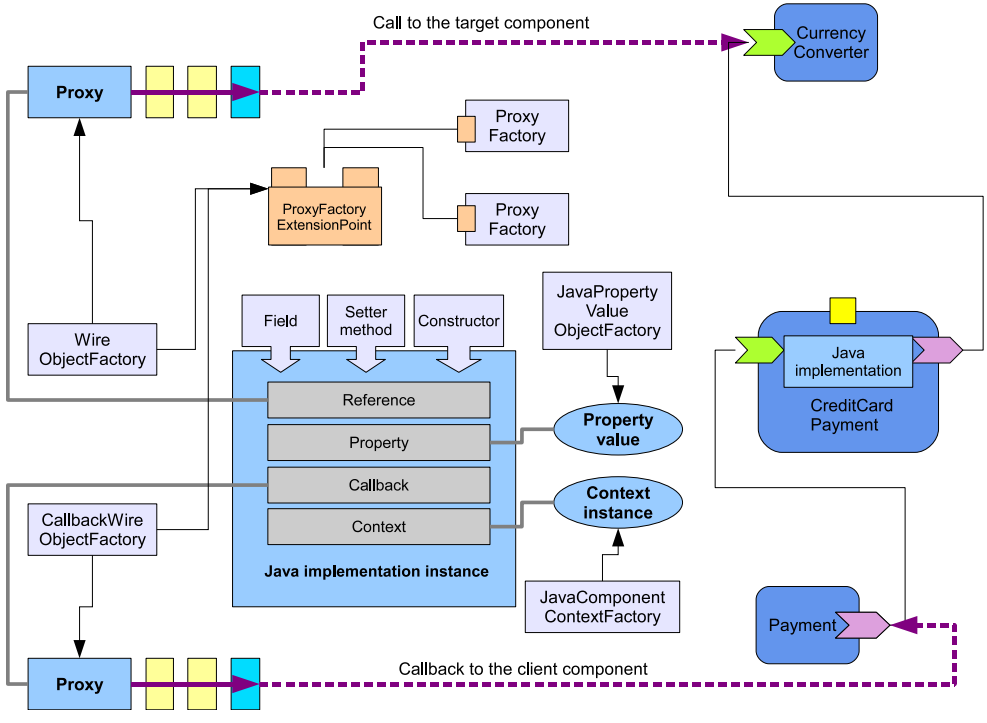


Figure 13.18 Dependency injection and proxy creation performed by JavaImplementationProvider for Java implementation classes

Stopping a component implementation destroys all the implementation instances and related resources.

STARTING AND STOPPING SCA SERVICES AND REFERENCES

Services and references of an SCA component are started using the configured bindings. The binding provider that's contributed by the binding extension is responsible for setting up the connections to the underlying protocol. The Tuscany core delegates the start or stop call to the binding provider. The providers typically respond to these lifecycle events by doing the following:

- *Starting the service binding*—Publish the SCA service to the underlying protocol as an endpoint to listen for requests from the protocol layer. This makes the SCA service available to clients that support the binding protocol.
- *Starting the reference binding*—Prepare the SCA reference to make outbound calls to its service providers via the binding protocol.
- *Stopping the service binding*—Remove the corresponding endpoint from the protocol stack and release associated resources.
- *Stopping the reference binding*—Release connections to the protocol stack's associated resources.

We'll further discuss the responsibility of binding providers and illustrate how binding providers implement the lifecycle methods in chapter 14.

13.3.6 Invoking SCA references and services

Each binding configures the access method to be used when invoking a reference or a service. For example, the Web Services binding dictates that messages will be formatted as XML-based SOAP envelopes passing over HTTP or the JMS protocol. Let's look at how invocations are handled, starting with references.

INVOKING A SERVICE THROUGH AN SCA REFERENCE

SCA components can invoke other services by calling the service proxy of the corresponding reference binding. The service proxy can be looked up from the context or injected by the component implementation provider. The proxy is a façade of the component reference invocation chain. This is demonstrated in figure 13.19.

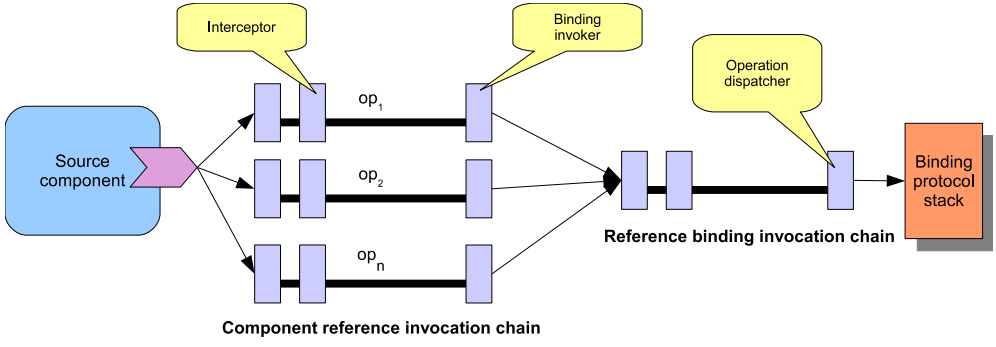


Figure 13.19 The invocation chain for an SCA reference binding

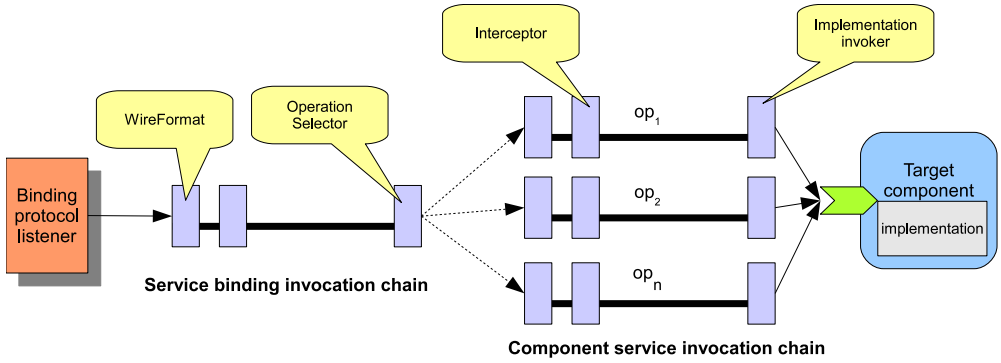


Figure 13.20 The invocation chains for an SCA service binding

When a method of the reference proxy is invoked, a Tuscany message is created to hold the input parameters. The interceptors in the invocation chain will be called one by one to process the message. At the end of the invocation chain, the binding `Invoker` calls the APIs from the binding protocol stack to invoke the service provider using messaging or RPC technologies.

RESPONDING TO INCOMING INVOCATIONS ON SCA SERVICES

SCA and non-SCA clients can invoke an SCA service using the access method configured by the binding. The request is picked up by a listener that the binding registers with the underlying protocol stack at start time, for example, a servlet listener or JMS MessageListener. Figure 13.20 shows the invocation chains that connect the service binding to the component implementation.

When the binding listener is called by the protocol stack, it creates a Tuscany `Message` from the protocol-specific message. The Tuscany `Message` is then passed to interceptors in the binding invocation chain. Among them is the wire format interceptor that'll decode and extract the input parameters from the protocol message. The operation selector will map the request to an operation (method request) in the service interface. Interceptors are invoked one by one to perform tasks such as transforming data or enforcing policies. At the end of the invocation chain, the implementation `Invoker` is called to dispatch the invocation to an instance of the component implementation to handle the business logic. The `Invoker` from the implementation extension will decide how to locate or instantiate an implementation instance so that the business logic can be invoked.

The response from the business logic, or any exceptions, will be passed back to the interceptors and `Invoker`s within the invocation chain. When it reaches the binding listener, the response is given to the binding protocol stack, which in turn sends it back to the client.

COMPONENT IMPLEMENTATION INSTANCE MANAGEMENT

SCA applications can choose whether implementation instances (such as a Java object) are shared across multiple invocations in a boundary called the component

scope. For example, the Java implementation type supports `STATELESS`, `COMPOSITE`, and `CONVERSATION` scopes. The implementation `Invoker` dispatches an invocation to an implementation instance within the scope. When required, an implementation provider is able to create an implementation instance for a component, appropriately initialize it before any request is accepted, and destroy it after it's no longer used. Tuscany uses the `ScopeContainer` to help manage this component instance lifecycle.

When a component implementation is activated, the `CompositeActivator` checks to see if it needs instance management. If it does, the `CompositeActivator` creates a `ScopeContainer` for the given component. The `ScopeContainer` holds the implementation instances and manages their creation and destruction depending on the scope of the component in question.

In this section we've talked in a fair amount of detail about the sequence of steps the Tuscany runtime performs in order to process a contribution's composite files. The loading, resolving, building, activating, and starting of a composite gives rise to an in-memory executable model of the composite's components. Incoming messages arriving at the Tuscany runtime, or indeed passing between components within the Tuscany runtime, are transformed and directed to the correct component using this in-memory model. When it comes time to shut down the runtime, the stop-and-deactivate step reverses the process and releases all of the resources held by the runtime model.

13.4 Summary

You now have a good understanding of the Tuscany architecture and are equipped with the knowledge to either extend Tuscany with new functionality or extend your own software with Tuscany's capability to process SCA applications. This chapter provided an in-depth view of Tuscany's modular architecture and the characteristics that enable Tuscany to be easily extended. We also discussed how Tuscany's lightweight runtime can be tailored to address specific application-processing requirements and therefore keep a small footprint.

A major portion of this chapter focused on demonstrating how the extensible infrastructure enables you to add support for new technologies that may not be already in Tuscany. You can leverage this knowledge now and add new implementation types or binding types in the next chapter.

14 Extending Tuscany

This chapter covers

- Building Tuscany extensions
- A complete walk-through of developing a new implementation type
- A complete walk-through of developing a new binding type

Apache Tuscany already supports a wide variety of technologies out of the box. But it's still possible that the technology that you're interested in isn't supported yet. The beauty of Tuscany is that it can be extended easily to support new technologies because of its modularized and pluggable architecture.

This chapter is somewhat different from the previous chapters in the book because it goes beyond describing how to use the Tuscany runtime to build applications and explains how to extend the Tuscany runtime to support new technologies. The nature of this subject means that this chapter's content is more complex than that of previous chapters. If you don't need to extend Tuscany, you can skip this chapter. If you do want to add your own extension, we'll present a practical step-by-step process for doing that, and we've based our examples on samples that come with the Tuscany distribution.

We'll start this chapter with a high-level view of what's required to extend Tuscany. Then we'll walk through two examples in detail. In one example we'll extend Tuscany to allow users to create a new Java-based implementation type called `implementation.pojo`. In the second example we'll extend Tuscany to support a new communication protocol called Echo with `binding.echo`. We won't cover the details of adding a databinding extension or a policy extension in this chapter. But the steps required for adding these extension types are similar to the two that we've chosen to demonstrate, and there are plenty of example databindings and policies in the Tuscany code base.

Let's start with an overview of the steps required to extend Apache Tuscany.

14.1 The high-level view of developing a Tuscany extension

The first step in supporting a new technology in Tuscany is to determine how it's relevant to building an SCA composite. Is it going to be used to build components, to handle a communication protocol, to describe a service or reference interface, to represent data, or to describe a quality of service? In other words, which extension type does it fit into? Tuscany defines extension types for each purpose, namely:

- Implementation types
- Binding types
- Interface types
- Databindings
- Intents and policy sets

The SCA specifications define how each extension type (with the exception of databinding, which is Tuscany-specific) can be used to enable SCA to work with a new technology. This is referred to as the SCA extension model. When you want to support a new technology in Tuscany, you build a new extension for an appropriate extension. The SCA specifications themselves use this extension model to describe a series of extensions including `binding.ws` (Web Services), `binding.jms` (JMS), `implementation.java` (Java), `implementation.spring` (Spring), and `implementation.bpel` (BPEL). Tuscany supports these and in addition implements other extensions such as `implementation.script` (scripting languages) and `binding.atom` or `binding.jsonrpc` (Web 2.0).

Creating a new extension in Tuscany involves two distinct steps. First, you'll need to develop the extension code to handle the new technology. For example, `implementation.java` is implemented using reflection APIs to invoke each business method. We won't cover the details of this step in this chapter because each extension is dependent on the characteristics of the technology that it supports. In the second step, the Tuscany runtime is configured to load, invoke, and manage the new extension. You'll provide this information to the Tuscany runtime through the Tuscany extension point mechanism.

An extension point is the place where the Tuscany runtime collects the information required for handling an extension. You'll need to do the following:

- Define how the extension can be used and configured in a composite file.
- Define how to create a runtime instance of the configured extension.
- Enable the Tuscany runtime to invoke and manage the extension.

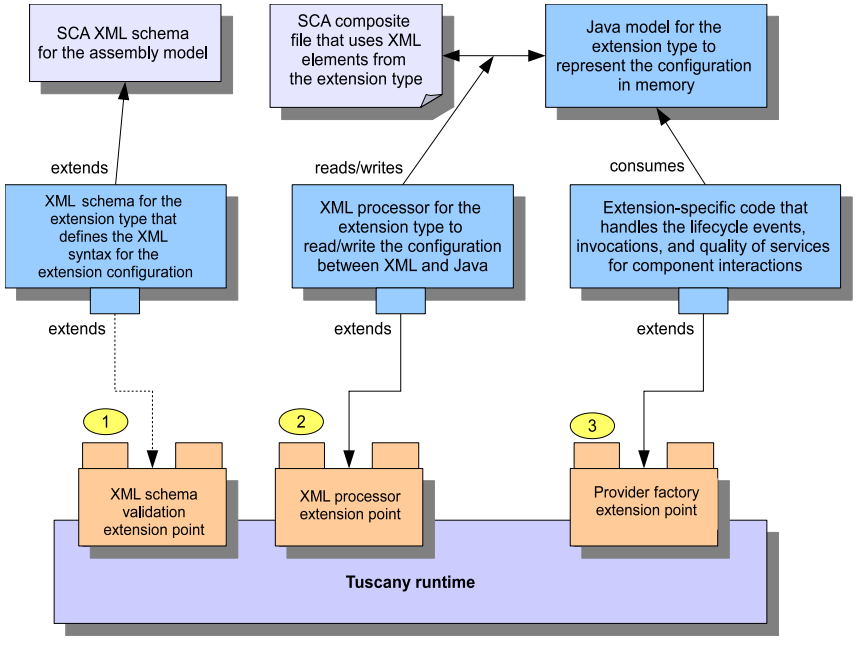


Figure 14.1 An extension contributes various information to Tuscany extension points.

Figure 14.1 shows the three typical extension points where you'll need to provide information to enable a new extension. Extension points are shown as plug-in boxes numbered 1, 2, and 3 on top of the runtime.

The numbered extension points from figure 14.1 are as follows:

1. *The extension point where you'll provide the XML schema for the extension*—This defines how the extension is used in the XML of an SCA composite file. Notice how it extends the schema that SCA defines for extension types. For example, `implementation.java` is an extension of the SCA implementation type and adds the `class` attribute for the name of the Java file that implements a component.
2. *The extension point where you'll define a Java model that represents the in-memory version of the extension*—You'll also provide the code for a processor that knows how to transform the XML representation in the composite file into an in-memory Java model and vice versa.
3. *The extension point where you'll add the code that the Tuscany runtime will use to locate, invoke, and manage the extension at runtime*—This is called an extension provider. For example, the BPEL implementation extension provider delegates the handling of the BPEL process component implementation to the Apache ODE runtime.

Now that you have a general understanding of the steps required for adding a new extension, let's look at some concrete examples. We'll first look at what's required to add a new implementation type.

14.2 Developing a POJO implementation type

In this section we'll show you how to develop a new implementation type. If, say, the TuscanySCATours company wanted to implement its Payment component using Fortran, Pascal, Cobol, or any one of many programming environments that the Tuscany SCA Java runtime doesn't support out of the box, a new implementation type would be needed.

Once the new implementation type is provided, anyone wanting to use it could exploit it by adding the appropriate `<implementation/>` element to their component description. One person takes the time to work out how to integrate with the new component implementation environment, and other users can benefit.

Here we'll add support for a POJO (plain old Java object) component implementation type, which can be used in the following way.

```
<component name="HelloWorldComponent">
    <p:implementation.pojo class="helloworld.HelloWorldImpl" />
</component>
```

This extension is a simplistic version of `implementation.java`, but we've chosen it because it's provided in Tuscany as a simple example to show how to add a new implementation type. You can find the code for the `implementation.pojo` extension, and examples of its use, in the Tuscany source or binary distribution in the samples/implementation-pojo-extension directory.

Let's start by looking at how to add XML schema for the `implementation.pojo` element.

14.2.1 Add the implementation.pojo XML schema

The XML schema for each implementation extension defines how it will appear within the XML of an SCA composite file. This information resides in an XSD file that's typically prefixed with the extension name. The following listing shows the contents of sample-implementation-pojo.xsd.

Listing 14.1 XML schema for `implementation.pojo`

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://pojo"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:p="http://pojo"
    elementFormDefault="qualified">
    <import namespace="http://www.osoa.org/xmlns/sca/1.0"/>

<element name="implementation.pojo"
    type="p:POJOImplementation"
    substitutionGroup="sca:implementation"/>

    <complexType name="POJOImplementation">
        <complexContent>
            <extension base="sca:Implementation">
                <attribute name="class" type="NCName" use="required"/>
```

Define implementation.pojo type

```
            </extension>
        </complexContent>
    </complexType>
</schema>
```

Define implementation.pojo type

The content of this .xsd file describes some key attributes of the `implementation.pojo` extension. It declares a global element, `implementation.pojo`, that belongs to the `"http://pojo"` namespace. The `<p:implementation.pojo>` element can be used to substitute `<sca:implementation>` as a child of the `<sca:component>` element. The `POJOImplementation` type extends `<sca:Implementation>` by adding a `class` attribute of type `NCName`.

The Tuscany runtime will use the schema for the extension to validate the `implementation.pojo` syntax as it reads the composite file into memory. The validation will be done only if the schema is registered with the Tuscany runtime.

You'll want the `implementation.pojo` extension to be validated in this case. Alongside your new XML schema file you register the schema with the Tuscany runtime by creating a file under the META-INF/services directory as follows:

```
src/main/resources
    META-INF/services/
        org.apache.tuscany.sca.contribution.processor.ValidationSchema
    sample-implementation-pojo.xsd
```

This ValidationSchema file will contain a single line to identify which schema to register:

```
    sample-implementation-pojo.xsd
```

Schema validation is an optional step in forming an extension. It's highly recommended, though, because it allows the Tuscany runtime to report potential problems early on.

Now that you have the XML schema, you'll need to define how the `implementation.pojo` XML element is read in and written out.

14.2.2 Adding implementation.pojo XML processing

Before you read and write the implementation XML, you'll first need to define how the implementation is modeled in the Tuscany runtime. In figure 14.2, box 1, we'll introduce `implementation.pojo` to our composite application to implement the HelloWorld component. The `class` attribute holds the name of the business logic for the HelloWorld component, in this case helloworld.HelloWorldImpl.

> **SCA assembly naming convention for implementation types**
>
> The SCA assembly spec has a naming convention for the XML element of an implementation type. The local name uses the "implementation" prefix, for example, `implementation.pojo`. The target namespace for elements not defined by the SCA specifications must be defined in namespaces other than the SCA namespace (http://www.osoa.org/xmlns/sca/1.0). We'll use http://pojo for `implementation.pojo`.

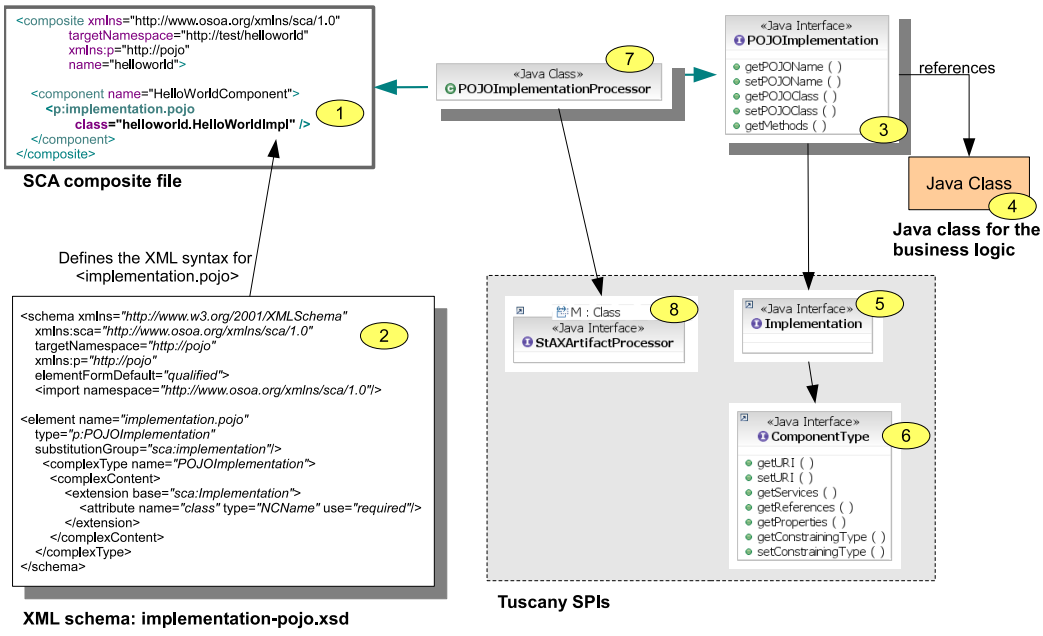


Figure 14.2 The XML and Java model for the POJO implementation type

Figure 14.2, box 2 holds the XML schema for `implementation.pojo` that we defined earlier. You'll use this to validate the XML configuration.

The next step is box 3, where you'll define the in-memory Java model of `implementation.pojo`. The `<implementation.pojo>` element is modeled as a pure Java interface and extends the SCA `Implementation` interface. The following code snippet shows the key methods we defined:

```
import org.apache.tuscany.sca.assembly.Implementation;

public interface POJOImplementation extends Implementation {
    public String getPOJOName();
    public void setPOJOName(String pojoName);
    public Class<?> getPOJOClass();
    public void setPOJOClass(Class<?> pojoClass);
    public Map<String, Method> getMethods();
}
```

The `POJOName` attribute will be set to the name from the `class` attribute, `helloworld.HelloWorldImpl` in our example. This will later on be linked to the `helloworld.HelloWorldImpl` class (shown as box 4) during the model-resolution phase.

To create instances of the `implementation.pojo` model you'll need a factory, as shown in the following snippet:

```
public interface POJOImplementationFactory {
    POJOImplementation createPOJOImplementation();
}
```

The Java model factory for `implementation.pojo` must be registered with the Tuscany runtime. Add the model class and factory and the META-INF/services POJOImplementationFactory file to your list of files:

```
src/main/java/
    pojo/
        impl/
            POJOImplementationFactoryImpl.java
            POJOImplementationImpl.java
        POJOImplementation.java
        POJOImplementationFactory.java

src/main/resources/
    META-INF/services/
        pojo.POJOImplementationFactory
```

The POJOImplementationFactory file contains a single line as follows:

```
pojo.impl.POJOImplementationFactoryImpl
```

Let's check our to-do list. By now we have the XML model, the Java model, and the capability to create `implementation.pojo` models through the factory. The next step is to define a StAX-based processor that can read the XML and map it to the Java model and vice versa. This is shown in figure 14.2, box 7.

StAX: the Streaming API for XML

The Streaming API for XML (StAX) is a Java-based API for pull-parsing XML. It's defined by JSR 173: http://jcp.org/en/jsr/detail?id=173.

The following code snippet shows the detail of the StAX processor for `implementation.pojo`. It extends the `org.apache.tuscany.sca.contribution.processor.StAXArtifactProcessor` interface. Two methods provide the keys for looking up the processor. The `getArtifactType()` method returns the XML QName for `<implementation.pojo>`. The `getModelType()` method returns the `POJOImplementation` Java model class. Using these keys, Tuscany can find the right processor when it comes across `<implementation.pojo/>` when reading a composite file and when it comes across `POJOImplementation` when writing out the in-memory model to XML.

```
public class POJOImplementationProcessor implements
     StAXArtifactProcessor<POJOImplementation> {
   private static final QName IMPLEMENTATION_POJO
           = new QName("http://pojo", "implementation.pojo");
    ...
    public QName getArtifactType() {
        return IMPLEMENTATION_POJO;
    }

    public Class<POJOImplementation> getModelType() {
        return POJOImplementation.class;
```

```
    }
    ...
}
```

The `POJOImplementationProcessor` needs to access other classes in the Tuscany extension point registry such as the `POJOImplementationFactory`. The constructor of the `POJOImplementationProcessor` includes the `ModelFactoryExtensionPoint` as an argument to make this possible. This is shown in the following code snippet. The Tuscany runtime will inject an instance of the model factory extension point from the extension registry.

```
public POJOImplementationProcessor(ModelFactoryExtensionPoint
                                   modelFactories, Monitor monitor) {
     ...
     pojoImplementationFactory =
                 modelFactories.getFactory(POJOImplementationFactory.class);
     ...
}
```

Now you're ready to provide the code to read the XML model and to create its corresponding Java model. The code sample for the `read()` method of `implementation.pojo` is shown here:

```
public POJOImplementation read(XMLStreamReader reader)
                     throws ContributionReadException, XMLStreamException {
    POJOImplementation implementation =
                     pojoImplementationFactory.createPOJOImplementation();
    ...
    String className = reader.getAttributeValue(null, "class");
    implementation.setPOJOName(className);

    ...
    return implementation;
}
```

This code reads the `class` attribute from the `<implementation.pojo/>` element.

You'll also need to provide a `write()` method whose purpose is to create XML output based on a Java model. The code snippet for `implementation.pojo` is as follows:

```
public void write(POJOImplementation implementation,
                  XMLStreamWriter writer)
            throws ContributionWriteException, XMLStreamException {
     writer.writeStartElement(IMPLEMENTATION_POJO.getNamespaceURI(),
                              IMPLEMENTATION_POJO.getLocalPart());

     if (implementation.getPOJOName() != null) {
        writer.writeAttribute("class", implementation.getPOJOName());
     }

     writer.writeEndElement();
}
```

You've finished implementing the StAX processor for `implementation.pojo`. Again, you'll need to register it with a Tuscany extension point. Add a POJOImplementation-

Processor and a StAXArtifactProcessor file to your collection of files under META-INF/services as follows:

```
src/main/java/pojo/impl/
    POJOImplementationProcessor.java
src/main/resources/
    META-INF/services/
        org.apache.tuscany.sca.contribution.processor.StAXArtifactProcessor
```

This StAXArtifactProcessor file contains a single line as follows:

```
pojo.impl.POJOImplementationProcessor;qname=http://pojo#implementation.pojo,
➥ model=pojo.POJOImplementation
```

We've now explained how to develop the XML model and the Java model and provided a StAX processor that transforms one form to another. Let's now look into how this information is used to create a component.

14.2.3 Determining the component type for implementation.pojo

Tuscany needs to be able to understand how to map the POJO, specified by `implementation.pojo`, to an SCA component's services, references, and properties.

So far the Tuscany runtime knows only the name of the class associated with `implementation.pojo`. In order to access the class object, the class name needs to be linked to a real POJO object. Although this seems like a simple step for POJO, the resolution of artifact names to real objects can be different and more complicated for each implementation type. Therefore, for each implementation type you'll need to provide code in the `resolve()` method of the model processor to enable the Tuscany runtime to find the necessary artifacts. The following listing shows how the `class` attribute for `implementation.pojo` is resolved to the class object through the ModelResolver.

Listing 14.2 The first half of the `implementation.pojo` processor `resolve()` method

```
public void resolve(POJOImplementation implementation,
                    ModelResolver resolver)
    throws ContributionResolveException {
        ClassReference classReference = new
                            ClassReference(implementation.getPOJOName());
        classReference = resolver.resolveModel(ClassReference.class,
                                               classReference);
        Class<?> pojoClass = classReference.getJavaClass();
        if (pojoClass == null) {
            throw new ContributionResolveException("Class not resolved: " +
                                            implementation.getPOJOName());
        }
        implementation.setPOJOClass(pojoClass);
        ...
}
```

The `resolve()` method creates a `ClassReference` object from the name of the POJO and passes it to the model resolver. The model resolver finds a `ClassReference` object from the contribution that matches the name of the POJO. The returned

`ClassReference` already has the Java class loaded. The method finally sets the `POJOClass` on the `POJOImplementation` model object.

Next, you'll need to create the component and find the constructs that represent services, references, and properties. The names for these attributes are either available in the componentType file or, depending on the underlying technology, can be introspected from the implementation itself. Let's see how this can be done for a POJO.

Listing 14.3 shows the second half of the `implementation.pojo resolve()` method. It uses the `POJOImplementation` model to determine if a componentType file exists. If so, you'll use the information available in the componentType file to create the component. Otherwise, you'll introspect the POJO implementation class and create a component based on the information found.

Listing 14.3 The second half of the `implementation.pojo` processor `resolve()` method

```
public void resolve(POJOImplementation implementation,
ModelResolver resolver) throws ContributionResolveException {
   ...
    ComponentType componentType = assemblyFactory.createComponentType();
 componentType.setUnresolved(true);
 componentType.setURI(implementation.getURI() + ".componentType");
 componentType = resolver.resolveModel(ComponentType.class, componentType);

 if (!componentType.isUnresolved()) {
        implementation.getServices().addAll(componentType.getServices());
     implementation.getReferences().addAll(componentType.getReferences());
     implementation.getProperties().addAll(componentType.getProperties());
 } else {
     Service service = assemblyFactory.createService();
        service.setName(pojoClass.getSimpleName());
     JavaInterface javaInterface;
     try {
         javaInterface = javaFactory.createJavaInterface(pojoClass);
     } catch (InvalidInterfaceException e) {
         throw new ContributionResolveException(e);
     }
     JavaInterfaceContract interfaceContract =
javaFactory.createJavaInterfaceContract();
     interfaceContract.setInterface(javaInterface);
     service.setInterfaceContract(interfaceContract);
     implementation.getServices().add(service);
 }

 implementation.setUnresolved(false);
}
```

Notice that we're using the implementation URI to locate the component type file. It's expected to be found at `helloworld/HelloWorldImpl.componentType`. This URI is formed from the location of the class file by replacing the .class suffix `.componentType`.

You now have an in-memory representation of the component type for `implementation.pojo`. The next step is to provide a hook for the implementation type so that the Tuscany runtime can invoke, start, and stop implementation instances.

14.2.4 Controlling implementation.pojo invocation and lifecycle

Each component implementation type has an associated lifecycle that allows it to be started, invoked, and stopped. Tuscany defines an `ImplementationProvider` interface for this purpose as well as an `ImplementationProviderFactory` interface to create instances of the implementation provider. Each extension provides its own implementation of the `ImplementationProvider` and `ImplementationProviderFactory` interfaces. The class that implements the `ImplementationProviderFactory` interface for the given extension is registered with Tuscany.

The following code shows the `POJOImplementationProviderFactory` class, which creates `POJOImplementation` providers:

```
public class POJOImplementationProviderFactory implements
                        ImplementationProviderFactory<POJOImplementation> {

    public
        POJOImplementationProviderFactory(ExtensionPointRegistry registry){
    }

    public Class<POJOImplementation> getModelType() {
        return POJOImplementation.class;
    }

    public ImplementationProvider createImplementationProvider(
                                        RuntimeComponent component,
                                        POJOImplementation implementation) {
        return new POJOImplementationProvider(component, implementation);
    }
}
```

Notice that the constructor of this class has Tuscany `ExtensionPointRegistry` as a parameter. This is the source for finding any extension point, and therefore extension, that's registered with the Tuscany runtime. Extension types are indexed by their model class name in the registry. Therefore, the `getModelType()` method returns the `POJOImplementation` model class that's used by the Tuscany runtime to locate the provider factory.

You'll add the provider factory and register it by adding an ImplementationProviderFactory file to the list of files in META-INF/services:

```
src/main/java/
    pojo/provider/
        POJOImplementationProviderFactory.java
src/main/resources/
    META-INF/services/
        org.apache.tuscany.sca.provider.ImplementationProviderFactory
```

The content of the ImplementationProviderFactory file is a single line, as follows:

```
pojo.provider.POJOImplementationProviderFactory;
➥ model=pojo.POJOImplementation
```

You've now enabled the Tuscany runtime to create components using `implementation.pojo`. Next, we'll provide the code that handles the invocation, start, and stop of

the component instances by implementing the `ImplementationProvider` interface, as shown here.

Listing 14.4 The implementation provider for `implementation.pojo`

```
class POJOImplementationProvider implements ImplementationProvider {

    POJOImplementationProvider(RuntimeComponent component,
                               POJOImplementation implementation) {
        this.implementation = implementation;
        try {
            pojoInstance = implementation.getPOJOClass().newInstance();
        } catch (Exception e) {
            throw new ServiceRuntimeException(e);
        }
    }

    public void start() {
        ...
        Method initMethod = implementation.getMethods().get("init");
        if (initMethod != null) {
            initMethod.invoke(pojoInstance);
        }
    }

    public void stop() {
        ...
        Method destroyMethod =
             implementation.getMethods().get("destroy");
        if (destroyMethod != null) {
            destroyMethod.invoke(pojoInstance);
        }
    }

    public Invoker createInvoker(RuntimeComponentService service,
                                 Operation operation) {
        Method method = implementation.getMethods().
            get(operation.getName());
        POJOImplementationInvoker invoker =
            new POJOImplementationInvoker (pojoInstance,
                                           operation,
                                           method);
        return invoker;
    }
    ...
}
```

1 Start the implementation

2 Stop the implementation

3 Create an implementation.pojo invoker

In this case the provider constructor creates a single instance of the implementation class. Because there's one provider instance per component, a single instance of the POJO class is created for each component. Alternative implementation creation algorithms can be created as required.

The `start()` method 1 prepares the implementation artifacts for the extension; for example, compile the BPEL script for `implementation.bpel`, or handle dependency injection of references and properties for `implementation.java`.

The `stop()` method ❷ is called when a component is stopped. It provides the code to destroy the implementation instance and release its resources.

An instance of an invoker is required for each service operation. The Tuscany runtime calls the `createInvoker()` method ❸ on the `POJOImplementationProvider` to create an instance of the invoker that's shared for all requests to the given operation. The invoker takes the incoming message from the service binding and passes it on to the appropriate operation in the implementation instance.

As you've seen, an SCA service can offer multiple operations. The `invoke` method in the `Invoker` interface, shown in the following snippet, is responsible for dispatching requests to the appropriate operation. The handling of the `invoke` operation is dependent on the technology used. As you can see, you'll need to provide an invoker specific to POJO.

```
class POJOImplementationInvoker implements Invoker {
    ...
    public Message invoke(Message msg) {
        try {
            // Use java reflection call to invoke the method
            // and set the result to the message body
            msg.setBody(method.invoke(pojoInstance,
                                      (Object[])msg.getBody()));
        } catch (InvocationTargetException e) {
            msg.setFaultBody(e.getCause());
        } catch (Exception e) {
            throw new ServiceRuntimeException(e);
        }
        return msg;
    }
}
```

Add the provider and invoker files to your list of files in the `implementation.pojo` module:

```
src/main/java/
    pojo/provider/
        POJOImplementationInvoker.java
        POJOImplementationProvider.java
```

You've now completed the work of extending Tuscany to support the POJO implementation type. In the next section, let's look at how all the pieces that we talked about so far come together in an end-to-end picture.

14.2.5 The end-to-end picture for the POJO implementation type

Figure 14.3 demonstrates the relationship among the classes that together enable the `implementation.pojo` extension in the Tuscany runtime.

During the contribution-processing phase, the StAX processor sees the `<implementation.pojo>` element in the composite file (1). Tuscany looks up the `POJOImplementationProcessor` (3) from the `StAXArtifactProcessorExtensionPoint` (D) using the QName of `implementation.pojo`. The `POJOImplementationProcessor`

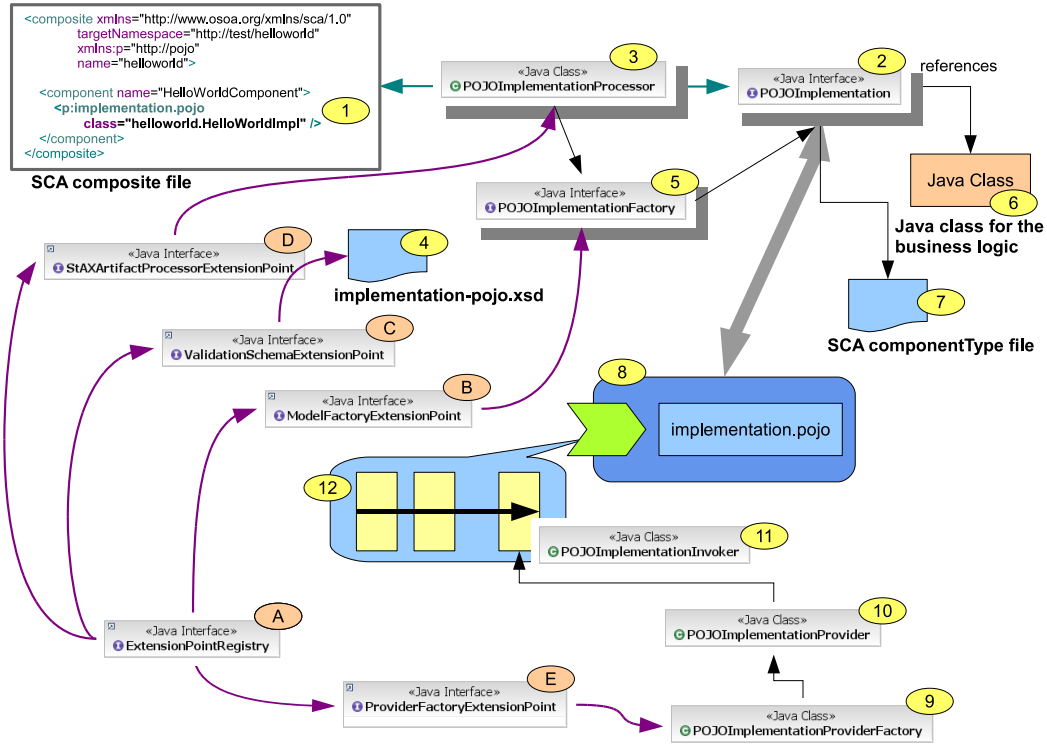


Figure 14.3 The classes that describe the extension point and extensions that collaborate to define the `implementation.pojo` extension

gets `POJOImplementationFactory` (5) from `ModelFactoryExtensionPoint` (B) and uses it to create an instance of `POJOImplementation` (2). The processor then parses the XML to populate `POJOImplementation`. If schema validation is enabled, the implementation-pojo.xsd file (4) is loaded by `ValidationSchemaExtensionPoint` (C), and it validates the `implementation.pojo` element as StAX parsing continues.

After the composite file is loaded, the `POJOImplementationProcessor` resolves the `POJOImplementation` object to get the Java class (6) and the accompanying componentType side file (7). Through the parsing of (7) and/or introspection of (6), it builds a component type (8) for the POJO implementation.

When the component is activated by the Tuscany runtime, the `ProviderFactoryExtensionPoint` (E) is queried using the model class (2), and `POJOImplementationProviderFactory` (9) is found. The Tuscany runtime calls the `POJOImplementationProviderFactory` to create an instance of the `POJOImplementationProvider` (10) for each component implementation instance. The `POJOImplementationProvider`'s `start()` method is invoked to set up the implementation instance when the component is started. The Tuscany runtime also calls the `POJOImplementationProvider`'s `createInvoker()` method to create an `Invoker` (11) for each service operation and add it to the invocation chain (12). When the request comes in, the

`POJOImplementationInvoker` will be called to dispatch the request to the POJO instance and get the response back to the caller. When the component is stopped, the `POJOImplementationProvider`'s `stop()` method is triggered to clean up the resources associated with the implementation instance.

Now that you have a good understanding of how all the pieces work together, let's package `implementation.pojo` and make it available as an implementation type.

14.2.6 Packaging the POJO implementation type

In order to make `implementation.pojo` available to Tuscany users, we'll need to package the extension into a JAR file and include it in the lib folder of the Tuscany distribution. The final package will contain the resources and Java classes summarized in table 14.1.

Table 14.1 The artifacts required to implement our new `implementation.pojo` extension

Resource or Java file	Description
sample-implementation-pojo.xsd	The XML schema for `<implementation.pojo>`
META-INF/services/pojo.POJOImplementationFactory	Service provider file to register `POJOImplementationFactoryImpl`
META-INF/services/ org.apache.tuscany.sca.contribution.processor. StAXArtifactProcessor	Service provider file to register `POJOImplementationProcessor`
META-INF/services/ org.apache.tuscany.sca.contribution.processor. ValidationSchema	Service provider file to register sample-implementation-pojo.xsd for schema validation
META-INF/services/ org.apache.tuscany.sca.provider.Implementation ProviderFactory	Service provider file to register `POJOImplementationProviderFactory`
pojo/POJOImplementation.java	Java interface for POJO implementation model
pojo/POJOImplementationFactory.java	Java interface for the model factory
pojo/impl/POJOImplementationFactoryImpl.java	Implementation class for the factory
pojo/impl/POJOImplementationImpl.java	Implementation class for the model
pojo/impl/POJOImplementationProcessor.java	StAX processor for `<implementation.pojo>`
pojo/provider/POJOImplementationProviderFactory.java	Implementation provider factory for `implementation.pojo`
pojo/provider/POJOImplementationProvider.java	Implementation provider for `implementation.pojo`
pojo/provider/POJOImplementationInvoker.java	Implementation invoker for `implementation.pojo`

You may now wonder whether you can follow the same steps to add other extension types to Tuscany. The answer is yes, with some minor differences. Next, let's look at how you can add a new binding type to Tuscany.

14.3 Developing a new binding type

In this section we're going to show you how to develop a new binding type. If the TuscanySCATours company wanted to connect its Payment component to the CreditCardPayment component using SMTP, FTP, DCE, XML-RPC, or any one of many communication protocols that the Tuscany SCA Java runtime doesn't support out of the box, they'd have to write a new binding type. As with implementation types, the advantage of creating a new binding type is that anyone else who needs to use the same protocol can do so by configuring their SCA services or references with the new `<binding/>` element.

Here we're going to explain how to build a new binding extension to wrap protocol-specific code. We'll do this through the implementation of a simple binding that handles a protocol of our own invention called the echo protocol. For a component running in isolation, the echo binding returns the first parameter of any outgoing message. If a reference is wired to a service, then the echo binding will forward the message, and the service is free to return whatever it likes.

We're using the echo binding because the code for the `binding.echo` extension, and examples of its use, can be found in the Tuscany source or binary distribution in the samples/binding-echo-extension directory.

There are two distinct steps for adding a new binding type. First, you'll find some code that handles the network protocol for the new binding, which, to Tuscany, is a black box. For example, for `binding.ws`, Tuscany uses Apache Axis to handle the SOAP Web Services protocol. Then you'll enable the binding through Tuscany extension points.

We won't talk through the details of the protocol-specific code here, but there are some key questions that you'll need to think through when integrating the protocol-specific code with the binding extension. Think of the binding as a pipe that handles communication between SCA and non-SCA components. At one end it receives the message (inbound invocation request), and on the other end it delivers it (outbound invocation request). The following are the types of capabilities that the binding should be able to provide:

- *Dispatching an outbound SCA invocation request to the underlying protocol handler*—The dispatching can be done in different styles depending on what the protocol requires. You might use RPC APIs to invoke remote services or use messaging APIs to send the request and receive the response. For example, the Web Services reference binding maps an SCA request to a SOAP envelope, connects to the HTTP endpoint, sends the SOAP message, receives a SOAP message as response, extracts the data from the SOAP response, and gives it back to SCA.

- *Handling inbound invocations that are calls from the protocol stack to the service*—An SCA binding usually registers a listener with the underlying stack that's called to route a request to the SCA component. Examples of listeners include Servlet, for the HTTP binding, and JMSServiceListener, for the JMS binding.
- *Mapping the data format that's understood by an implementation type to the wire format required by the protocol stack and vice versa*—Each protocol has its own requirement for handling data on the wire; for example, RMI and EJB bindings require the data to be Java serializable. Some protocols already provide a well-defined wire format; for example, Web Services with SOAP binding uses SOAP envelopes. The Tuscany runtime handles data format transformation through its databinding extension type, as discussed in detail in chapter 9. The interface for the binding and component implementation type defines the data format for each side of the data transformation. The Tuscany runtime reads these interfaces to determine the data format required and uses the appropriate databinding to handle the transformation seamlessly. Tuscany provides a rich set of databindings and can be easily extended to support additional ones if needed.

Now you have a rough idea of what we're trying to achieve in creating a new binding extension. Let's get started with the first step of creating the binding schema.

14.3.1 Adding the binding.echo XML schema

When defining the XML model for a binding, you'll need to think about how the binding will be used. Is it going to be configured for references only, services only, or both? A binding configured for references defines how an SCA component communicates with a service provider via an SCA reference. For example, `binding.ws` enables an SCA component to call a web service over SOAP/HTTP. A binding configured for service defines how an SCA service exposes its business function to other services over a particular protocol. For example, an SCA service with `binding.ws` will be published as a web service, and it can then be accessed by a JAX-WS client, a .NET client, or an SCA component using the Web Services protocol.

In our example, `binding.echo` will be configured for references and services. We define the schema for `binding.echo` in a file called sample-binding-echo.xsd. The schema is optional, but we'll include it in this example so that the Tuscany runtime can use it to validate the syntax of `binding.echo` as it appears in the composite file. The following listing shows the XML schema for the `<binding.echo>` element.

Listing 14.5 XML schema for `binding.echo` defined in sample-binding-echo.xsd

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://echo"
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:e="http://echo"
    elementFormDefault="qualified">
```

```
    <import namespace="http://www.osoa.org/xmlns/sca/1.0"/>

    <element name="binding.echo" type="e:EchoBinding"/>

    <complexType name="EchoBinding">
         <complexContent>
            <extension base="sca:Binding">
           ... <!-- binding specific elements or attributes can go here -->
            </extension>
        </complexContent>
    </complexType>
</schema>
```

Because `binding.echo` is a simple binding, you'll only need to extend SCA's programming model to define a name and type for this binding. Add the schema file and register it with Tuscany's schema validation extension point, in the same way as you did with `implementation.pojo`, by adding a line to a ValidationSchema file in the META-INF/services directory. The two files to add are these:

```
src/main/resources/
    sample-binding-echo.xsd
    META-INF/services/
        org.apache.tuscany.sca.contribution.processor.ValidationSchema
```

The ValidationSchema file contains a single line, as follows:

```
sample-binding-echo.xsd
```

Now that you have the XML schema for `binding.echo`, you'll need to define how it can be configured in the composite file and used at runtime.

14.3.2 Adding the binding.echo XML processor

The XML processor is used to read, write, and resolve `<binding.echo>` elements found in a composite file. In figure 14.4, box 1, we'll extend the SCA programming model for binding types and introduce the `binding.echo` extension. The schema for the Echo binding that we defined in the previous section is shown in box 2. It will be used by the Tuscany runtime to validate the syntax used in box 1. Two Java interfaces are implemented for the Echo binding type. The first is the model interface in box 3. This interface extends the base `Binding` interface that's marked as box 5. The second is the factory interface shown in box 4. This is used to create instances of `EchoBinding`. Unlike `implementation.pojo`, where we constructed specific XML processors, here we'll use the Tuscany generic StAX processor shown in box 6 to handle the XML/Java conversions.

Notice the `uri` attribute in box 1. All binding types have a `uri` attribute that represents the address of the target service endpoint. In the case of `binding.echo` the `uri` is used in a local in-memory cache to locate the target service.

A more complicated binding will have other attributes. For example, `binding.ws` supports an attribute called `wsdlElement` to point to WSDL-based configuration for the binding.

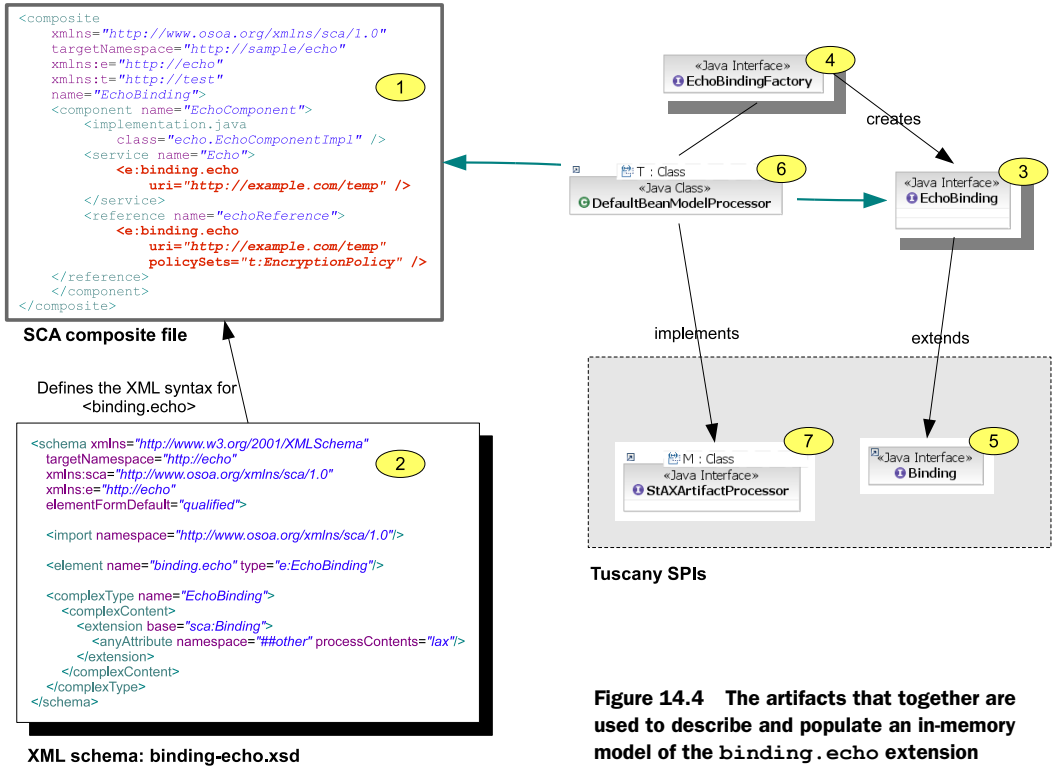


Figure 14.4 The artifacts that together are used to describe and populate an in-memory model of the `binding.echo` extension

Typically, the same model can be applied to both references and services. But it's possible for the model to be different depending on the type of the protocol supported by each. For example, `binding.jms` can use `connectionFactory` for references and `activationSpec` for services. These two attributes are mutually exclusive.

Let's look at the code from figure 14.4 in a little more detail starting with box 3. We'll provide a new Java interface specific to the Echo binding type that extends an existing Tuscany interface called `org.apache.tuscany.sca.assembly.Binding`. The code is shown in the following snippet:

```
package echo;
import org.apache.tuscany.sca.assembly.Binding;
public interface EchoBinding extends Binding {

}
```

We'll then provide a factory to create instances of the Echo binding type. This is shown in the following snippet. Notice that both the model and factory interfaces can be implemented using simple JavaBeans.

```
public interface EchoBindingFactory {
    EchoBinding createEchoBinding();
}
```

The Java model factory for `binding.echo` must be registered with the Tuscany runtime. Add the model class and factory and the META-INF/services echo.EchoBindingFactory file to your list of files:

```
src/main/java/
    echo/
        EchoBinding.java
        EchoBindingFactory.java
    impl/
        EchoBindingFactoryImpl.java
        EchoBindingImpl.java
src/main/resources/
    META-INF/services/
            echo.EchoBindingFactory
```

The META-INF/service echo.EchoBindingFactory file contains a single line as follows:

```
echo.impl.EchoBindingFactoryImpl
```

So far, we've created the XML and Java model for the Echo binding. We'll need to provide a StAX-based processor that bridges the two models. This is shown in box 6 of figure 14.4. The good news is that we don't have to write a new processor because the Echo binding is modeled as a JavaBean. Instead, we can use the `DefaultBeanModelProcessor` that Tuscany provides.

Using Tuscany's generic StAX processor

You can use Tuscany's generic StAX processor if the implementation type's XML model uses attributes of simple types and its Java model follows the JavaBeans pattern. Tuscany's generic StAX processor uses JavaBean property accessors to read/write the properties from/to the XML document. You can register the generic StAX processor with the following text (using your `qname/model/factory` names instead of CRUD):

```
org.apache.tuscany.sca.assembly.xml.DefaultBeanModelProcessor;qname=
http://
crud#implementation.crud,model=crud.CRUDImplementation,factory=crud.
CRUDImplementationFactory
```

We'll register the StAX artifact processor with the Tuscany runtime to enable it to handle the transformation between Java and XML models for the Echo binding. This is done by including a StAXArtifactProcessor file under META-INF/services as follows:

```
src/main/resources/
    META-INF/services/
        org.apache.tuscany.sca.contribution.processor.StAXArtifactProcessor
```

This file configures the generic Tuscany StAX processor using the following line:

```
org.apache.tuscany.sca.assembly.xml.DefaultBeanModelProcessor;qname=http://
     echo#binding.echo,model=echo.EchoBinding,factory=echo.EchoBindingFactory
```

Notice that the content of the file includes the processor class name called `DefaultBeanModelProcessor`. The `qname` attribute defines the XML QName of the binding element, `http://echo#binding.echo` in this case. The `model` attribute points to the class name of the binding's Java interface. The `factory` attribute is used to locate the EchoBindingFactory that's used to create instances of the configured Echo binding.

You've now completed the steps for defining the XML and Java model and providing a processor to handle the transformation. The next step is to enable the runtime to invoke and manage `binding.echo` instances.

14.3.3 Controlling binding.echo invocation and lifecycle

Each binding extension requires a provider factory that's responsible for creating a `ReferenceBindingProvider` to handle each outbound request and for creating a `ServiceBindingProvider` for handling each inbound request. Listing 14.6 shows the implementation of `EchoBindingProviderFactory`, where we'll provide the code for creating both reference and service providers. The Java model for the `binding.echo` extension is used to look up the provider factory in Tuscany's general `ProviderFactoryExtensionPoint`.

Listing 14.6 EchoBindingProviderFactory

```
public class EchoBindingProviderFactory implements
                                    BindingProviderFactory<EchoBinding> {

    private MessageFactory messageFactory;

    public EchoBindingProviderFactory(ExtensionPointRegistry
       extensionPoints){
            ModelFactoryExtensionPoint factories =
       extensionPoints.getExtensionPoint(ModelFactoryExtensionPoint.class);
       this.messageFactory = factories.getFactory(MessageFactory.class);
    }

    public ReferenceBindingProvider createReferenceBindingProvider(
                                    RuntimeComponent component,
                                    RuntimeComponentReference reference,
                                        EchoBinding binding) {
        return new EchoReferenceBindingProvider(component,
                                            reference,
                                            binding);
    }

    public ServiceBindingProvider createServiceBindingProvider(
                                        RuntimeComponent component,
                                        RuntimeComponentService service,
                                        EchoBinding binding) {
        return new EchoServiceBindingProvider(component,
                                            service,
                                            binding,
                                            messageFactory);
    }
```

```
    public Class<EchoBinding> getModelType() {
        return EchoBinding.class;
    }
}
```

The constructor method takes the `ExtensionPointRegistry` as a parameter. This gives the factory access to other extension points and therefore other extensions, such as the factory instance required to create a Tuscany `Message`.

You'll need to add the `EchoBindingProviderFactory` and register it with the Tuscany runtime. The new files you'll need are as follows:

```
src/main/java/
    echo/provider/
        EchoBindingProviderFactory.java
src/main/resources/
    META-INF/services/
        org.apache.tuscany.sca.provider.BindingProviderFactory
```

The BindingProviderFactory file in META-INF/service contains the following line:

```
echo.provider.EchoBindingProviderFactory;model=echo.EchoBinding
```

By now the Tuscany runtime knows how to create instances of `binding.echo` and has a provider factory for creating service and reference providers. Let's look at the providers in more detail.

CODING THE SERVICE BINDING PROVIDER FOR BINDING.ECHO

The next step is to implement the code that handles service requests for `binding.echo`. The two files you'll add are as follows:

```
src/main/java/
    echo/server/
        EchoServiceListener.java
    echo/provider/
        EchoServiceBindingProvider.java
```

The `EchoServiceBindingProvider` class is shown in the following listing.

Listing 14.7 EchoServiceBindingProvider

```
class EchoServiceBindingProvider implements ServiceBindingProvider {

    private RuntimeComponent component;
    private RuntimeComponentService service;
    private EchoBinding binding;
    private MessageFactory messageFactory;

    ...

    public InterfaceContract getBindingInterfaceContract() {
        return service.getInterfaceContract();
    }

    public boolean supportsOneWayInvocation() {
        return false;
    }
```

```
    public void start() {
        RuntimeComponentService componentService =
                                        (RuntimeComponentService)service;
        RuntimeWire wire = componentService.getRuntimeWire(binding);
        InvocationChain chain = wire.getInvocationChains().get(0);

        // Register with the hosting server
        String uri = binding.getURI();
        EchoServer.getServer().register(uri,
            new EchoServiceListener(chain.getHeadInvoker(),
                                    messageFactory));
    }

    public void stop() {
        String uri = component.getURI() + "/" + binding.getName();
        EchoServer.getServer().unregister(uri);
    }

}
```

The `EchoServiceBindingProvider` class implements a few methods that are essential for handling service requests for the Echo binding. These methods will be accessed by the Tuscany runtime.

- The `start()` method is invoked when a component service gets started. It creates and publishes the service endpoint to the underlying protocol stack. Typically the service binding provider registers a listener to the underlying protocol stack to receive the requests from the transport. Examples include Servlet for HTTP-based protocols or a JMSServiceListener for JMS. The listener is then responsible for dispatching the inbound call to the SCA service that's configured with the binding type. We don't reproduce the `EchoServiceListener` here because it's specific to the Echo protocol, but you can see it by looking at the sample in the Tuscany distribution.
- The `stop()` method is invoked when a component service is stopped. It'll ask the underlying stack to remove the endpoint and release resources associated with the service.
- The `getBindingInterfaceContract()` method returns the interface that represents the type of data used by the binding extension type. Based on this information and the target destination data format, the Tuscany runtime decides what kind of databinding to use to handle the transformation of data as it travels between the two points. For example, a Web Services binding based on Axis2 uses WSDL as the binding interface with an AXIOM databinding, whereas the corresponding component reference or service can use a Java interface with a JAXB or SDO databinding. This frees the binding code from complicated transformation code and allows it to handle data uniformly. Please refer to chapter 9 for more details on databinding.

- The `supportsOneWayInvocation()` method tells the Tuscany runtime whether the binding can support one-way invocation natively or needs the Tuscany runtime to handle asynchronous invocation. If help is needed, the Tuscany runtime will use a thread pool to schedule one-way operations.

If you recall, our Echo binding handles both service and references. We just explained how the service provider works for `binding.echo`. Let's look at the code for the reference provider.

CODING REFERENCE BINDING PROVIDER FOR BINDING.ECHO

The Echo binding needs a reference provider to handle outbound invocation requests. You'll add two more new files as follows:

```
src/main/java/
    echo/provider/
        EchoBindingInvoker.java
        EchoReferenceBindingProvider.java
```

The `EchoReferenceBindingProvider` class is shown here.

Listing 14.8 EchoReferenceBindingProvider

```
class EchoReferenceBindingProvider implements ReferenceBindingProvider {

    private RuntimeComponentReference reference;
    private EchoBinding binding;

    EchoReferenceBindingProvider(RuntimeComponent component,
                                 RuntimeComponentReference reference,
                                 EchoBinding binding) {
        this.reference = reference;
        this.binding = binding;
    }

    public Invoker createInvoker(Operation operation) {
        ...
        return new EchoBindingInvoker(binding.getURI());
    }

    public boolean supportsOneWayInvocation() {
        return false;
    }

    public InterfaceContract getBindingInterfaceContract() {
        return reference.getInterfaceContract();
    }

    public void start() {
    }

    public void stop() {
    }
}
```

The following methods in the `EchoReferenceBindingProvider` class will provide the supporting code to handle the outbound invocation. This code is invoked by the Tuscany runtime.

- The `start()` method is invoked by the Tuscany runtime when a component reference is started. This method allocates resources such as a connection pool needed for the outbound invocation. No extra resources are required by the Echo binding, and this method does nothing here.
- The `stop()` method is invoked by the Tuscany runtime when a component reference is stopped by the Tuscany runtime. It releases resources associated with the reference.
- The `createInvoker()` method is used by the Tuscany runtime to create an `Invoker` for the reference binding. This `Invoker` is added to the invocation chain of the SCA reference. More detail is provided in the next section.
- The `getBindingInterfaceContract()` method returns the interface that represents the data types supported by the binding. In the Echo binding case, it can return either `null` or the interface contract from the component reference because it doesn't require data transformations.
- The `supportsOneWayInvocation()` method tells the Tuscany runtime whether the binding can support one-way invocation natively without the Tuscany runtime's built-in asynchronous facility.

The `Invoker` is responsible for dispatching the outbound invocation requests for the given SCA reference to the protocol stack. It can be viewed as a bridge between the SCA world and the protocol handler. It receives the input parameters from the calling component reference in the form of an SCA message. It then uses protocol-specific APIs to dispatch the request and the input data to the protocol handler. Once the response comes back from the protocol layer, the `Invoker` extracts the output data into an SCA message that gets sent to the component implementation.

If you recall, a service can consist of multiple operations. In Tuscany, an instance of the binding `Invoker` is associated with each operation. The code for `EchoBindingInvoker` is shown in the following snippet:

```
class EchoBindingInvoker implements Invoker {
    private String uri;

    EchoBindingInvoker(String uri) {
        this.uri = uri;
    }

    public Message invoke(Message msg) {
        try {
            System.out.println("Passing thro invoker...");
            Object[] args = msg.getBody();
            Object result = EchoServer.getServer().call(uri, args);
            msg.setBody(result);
        } catch (Exception e) {
```

```
            msg.setFaultBody(e);
        }
        return msg;
    }
}
```

In the case of the Echo binding samples, the `Invoker` doesn't do anything particularly interesting. It invokes the `call()` operation on a service retrieved from the EchoServer. This has the effect of sending the message to the target service or, if there's no target service, of returning the first input parameter. In a real binding you're free to add any code you need here to interact with the protocol stack.

You've now completed the steps required to add the Echo binding. Now that you understand the basic building blocks, let's look at the other bindings in the Tuscany distribution to see how integration with real protocol stacks is performed.

In the next section, we'll look at the end-to-end picture of how all the pieces that we talked about relate to one another.

14.3.4 The end-to-end picture for the Echo binding type

Figure 14.5 provides an end-to-end view of all the artifacts required to create the Echo binding and the relationship of those artifacts with one another. The boxes identified with a letter represent a Tuscany extension point. The numbered boxes represent the code required to enable the extension points for a given binding type.

As shown in figure 14.5, the Tuscany extension point registry (A) maintains a list of all the extension points. The Echo binding type contributes code to all extension points (B, C, and D). Let's start with box 1 in the upper-left-hand corner.

During the contribution processing phase, the StAX processor sees the `<binding.echo>` element in the composite file (1). Tuscany looks up the `DefaultBeanModelProcessor` (3) from the StAXArtifactProcessorExtensionPoint (D) using the QName of `binding.echo`. The processor gets `EchoBindingFactory` (5) from ModelFactoryExtensionPoint (B) and uses it to create an instance of `EchoBinding` (2). The processor then parses the XML to populate `EchoBinding`. If schema validation is enabled, the sample-binding-echo.xsd file (4) is loaded by the ValidationSchemaExtensionPoint (C), and it validates the `binding.echo` element as StAX parsing continues.

When the component is activated by the Tuscany runtime, the ProviderFactoryExtensionPoint (E) is queried using the model class (2) and `EchoBindingProviderFactory` (7) is found.

If `binding.echo` is used to handle a reference, the Tuscany runtime calls the `EchoBindingProviderFactory` to create an instance of `EchoReferenceBindingProvider` (10) for the reference binding. The `EchoReferenceBindingProvider`'s `start()` method is invoked to set up the reference binding provider instance when the component is started. The Tuscany runtime also calls the `EchoReferenceBindingProvider`'s `createInvoker()` method to create an `Invoker` (11) for each operation and adds it to the invocation chain (13) for the reference. When the request comes in, the `EchoBindingInvoker` will be called to dispatch the request to the "echo" protocol layer and

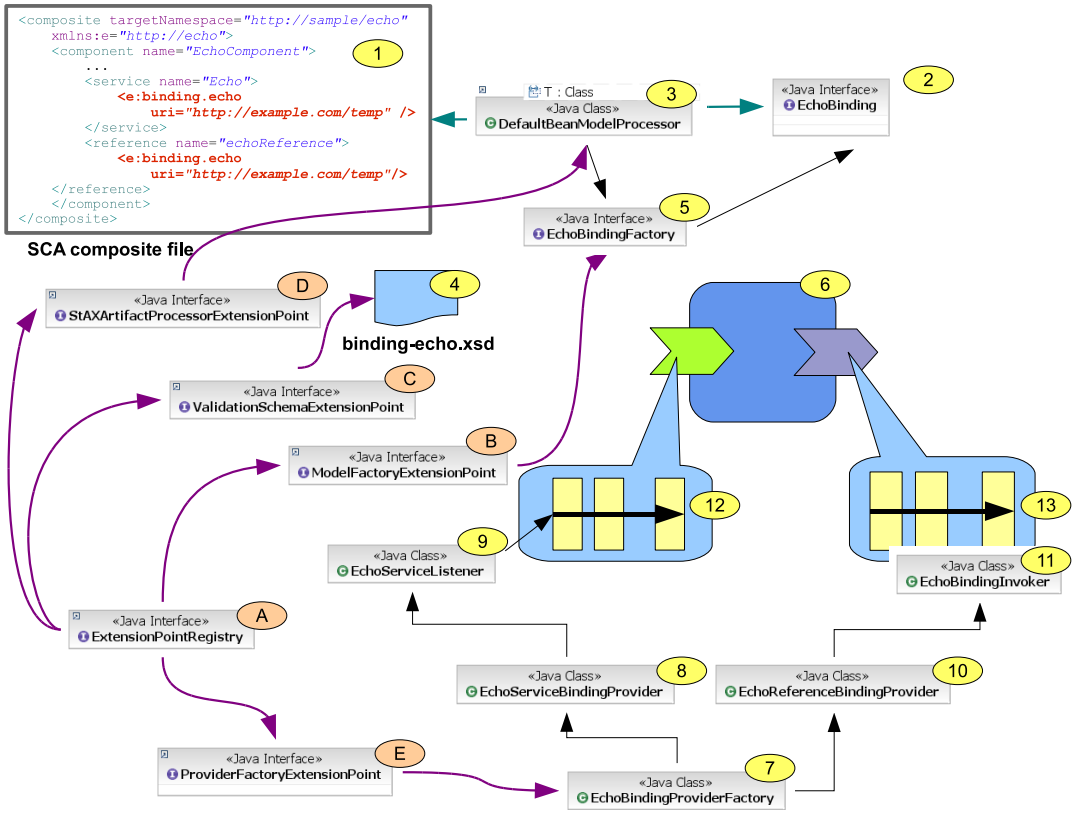


Figure 14.5 The different extension classes that collaborate and cooperate to support the our new binding type, `binding.echo`

get the response back to the caller. When the component is stopped, the `EchoReferenceBindingProvider`'s `stop()` method is triggered to clean up the resources associated with the reference instance.

If `binding.echo` is used to handle a service, the Tuscany runtime calls the `EchoBindingProviderFactory` to create an instance of `EchoServiceBindingProvider` (8) for the service binding. The `EchoServiceBindingProvider`'s `start()` method is invoked to create an `EchoServiceListener` (9) when the component is started. When a request comes from the "echo" protocol layer, the `EchoServiceListener` is triggered. It extracts the input data from the underlying message and dispatches to the head invoker of the invocation chain (12) for the service. When the component is stopped, the `EchoServiceBindingProvider`'s `stop()` method is triggered to clean up the resources associated with the service instance, including removing the `EchoServiceListener` from the "echo" protocol layer.

Now that you have a good understanding of how all the pieces work together to enable Echo binding in Tuscany, let's package `binding.echo` and make it available as a binding type.

14.3.5 Packaging the echo binding type

Package the Java classes and resource files for the `binding.echo` extension into a JAR file and include it in the lib folder of the Tuscany distribution. The final package is shown in table 14.2.

Table 14.2 The artifacts required to implement your new `binding.echo` extension

Resource or Java file	Description
META-INF/services/echo.EchoBindingFactory	Service provider file to register the model factory for `EchoBinding`
META-INF/services/org.apache.tuscany.sca.contribution.processor.StAXArtifactProcessor	Service provider file to register the StAX processor for `<binding.echo>`
META-INF/services/org.apache.tuscany.sca.contribution.processor.ValidationSchema	Service provider file to register sample-binding-echo.xsd for schema validation
META-INF/services/org.apache.tuscany.sca.provider.BindingProviderFactory	Service provider file to register the binding provider factory for `binding.echo`
sample-binding-echo.xsd	XML schema for `<binding.echo>`
echo/EchoBindingFactory.java	Java interface for the factory to create `EchoBinding`
echo/EchoBinding.java	Java interface for the `binding.echo` model
echo/impl/EchoBindingImpl.java	Implementation class for `EchoBinding`
echo/impl/EchoBindingFactoryImpl.java	Implementation class for the factory
echo/provider/EchoBindingInvoker.java	Binding `Invoker`
echo/provider/EchoReferenceBindingProvider.java	Reference binding provider
echo/provider/EchoServiceBindingProvider.java	Service binding provider
echo/provider/EchoBindingProviderFactory.java	Binding provider factory for `binding.echo`
echo/server/EchoServer.java	Mocked Echo server
echo/server/EchoServiceListener.java	Mocked Echo listener

In this section, we showed you how to add a new binding type to Tuscany by using a simple Echo sample. Any user who has access to this new `binding.echo` extension can extend their composite application descriptions to include the `<binding.echo/>` element.

You can use the techniques shown here of defining a model and creating processors and providers to build your own binding extensions. Tuscany has plenty of binding examples that you can refer to for inspiration, for example, Web Services, JMS, EJB, RMI, CORBA, JSON-RPC, and more.

We encourage you to think about contributing any new binding, or implementation, extensions back to the Tuscany community. In that way the Tuscany SCA runtime increases its coverage of supported binding and implementation technologies and becomes more attractive to a wider group of users.

14.4 Summary

SCA's extensible programming model offers the freedom of choice for selecting any technology for developing components, handling data, and handling communication protocols and policies.

In this chapter, you learned that Apache Tuscany's implementation of SCA is backed by a flexible, modular architecture that can be extended to support any technology. Adding support for a new technology requires you to determine how the technology is used to form SCA composites. This tells you which one of the supported extension types, such as implementation type and binding type, the new extension belongs to. You also learned that whereas the SCA assembly specification describes a specific extension configuration for implementations, interfaces, policy, and bindings, Apache Tuscany provides additional extension types that enable you to add support for new databindings.

We looked, at a high level, at what is required to add a new extension, and we studied the steps required to create a POJO implementation type and an Echo binding type. In doing so we presented the key interfaces and methods required for enabling an extension type through Tuscany's extension point mechanism. You're now ready to create new bindings or implementation types yourself.

Tuscany already supports many different extension types. But it's always possible that some technology isn't supported. As you go through the process of adding a new extension type, we encourage you to share your thoughts and to contribute the extension to the Tuscany project so that other members of the Tuscany community can benefit.

appendix A
Setting up

This chapter is a hands-on guide for getting you started with Tuscany and the examples that are used in this book. Here we'll talk about how to install Tuscany and set up the environment necessary to run and test the book examples and to develop your own SCA applications. We also give a brief overview of the Tuscany project so that you can navigate the website and interact with the Tuscany community.

A.1 Installing Tuscany

Installing Tuscany is simple: Download a Tuscany release and a Java JDK. The JDK is used to compile and run Java programs. Let's get started by first getting the Tuscany prerequisites.

A.1.1 Getting Tuscany's prerequisites

Running Tuscany SCA Java 1.x is dependent on JDK version 5.0 or later. The Tuscany samples can be run out of the box with Apache Ant and Apache Maven. To set up a development environment and run samples, follow these steps:

1. Download and install Java Development Kit (JDK) version 5 or later.
2. Download and install Apache Ant version 1.7.1 or later.
3. Download and install Apache Maven version 2.0.7 or later.

Now that you have the Tuscany prerequisites in place, you'll install Tuscany itself.

A.1.2 Downloading and installing Tuscany

Download Tuscany SCA Java 1.6 (or a later 1.x release) from the following web page: http://tuscany.apache.org/sca-java-1x-releases.html. This page contains the binary and source distributions of Tuscany SCA Java. For now, download the binary distribution for your platform. This contains a compiled version of the Tuscany core and extension JARs that are ready to be run. At a later point, if you want to

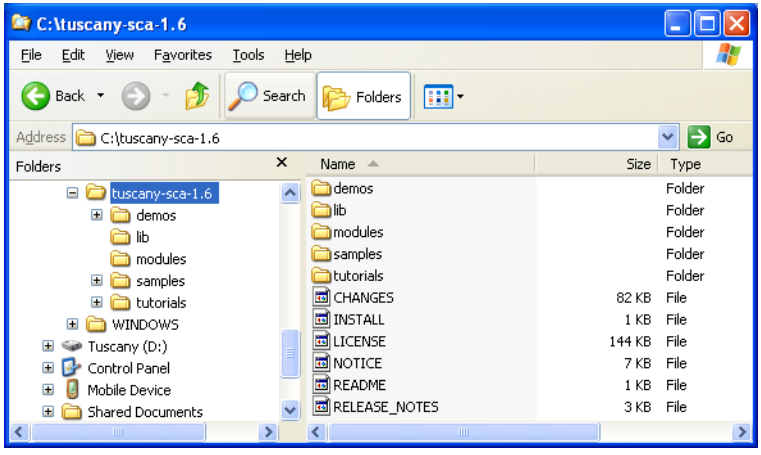


Figure A.1 The expanded Tuscany binary distribution on Windows

examine the Tuscany source code or make changes to it, you can download the source code distribution that contains all the Tuscany source code.

Once you've downloaded the Tuscany binary distribution, you'll need to expand it. Figure A.1 shows the expanded Tuscany binary distribution when unpacked to the C:\ directory on Windows.

The Tuscany binary distribution contains the directories shown in table A.1.

Table A.1 The contents of the Tuscany binary distribution

Path	What it contains
/modules	The Tuscany core and extension JARs
/lib	The third-party software that Apache Tuscany requires, and a manifest JAR that pulls together the third-party dependencies and Tuscany module JARs to make constructing the runtime classpath straightforward
/samples	Samples that demonstrate some of the features of Tuscany and SCA
/demos	More complex applications that bring together Tuscany features into working applications
/tutorials	An example based on an e-business website selling fruit and vegetables

The binary distribution doesn't include Javadoc. There's some Javadoc available on the Tuscany website at http://tuscany.apache.org/doc/javadoc/.

Now that you've expanded the binary distribution, let's test the Tuscany installation.

A.1.3 Testing the Tuscany installation

It's a good idea to use a simple sample to verify that Tuscany works properly in your environment before moving on to developing applications. The calculator sample is ideal for this because it's a very simple SCA application.

The calculator sample can be found in the samples/calculator directory. To run the sample with Ant, change into this directory and type

```
ant run
```

Assuming that Apache Ant has been installed correctly and is on your path, you'll see output similar to the following:

```
C:\tuscany-sca-1.6\samples\calculator>ant run
Buildfile: build.xml

run:
     [java] 14-Jan-2010 19:34:28 org.apache.tuscany.sca.node.impl.NodeImpl
         <init>
     [java] INFO: Creating node: Calculator.composite
     [java] 14-Jan-2010 19:34:29 org.apache.tuscany.sca.node.impl.NodeImpl
         configureNode
     [java] INFO: Loading contribution: file:/C:/tuscany-sca-1.6/
         samples/calculator/target/sample-calculator.jar
     [java] 14-Jan-2010 19:34:31 org.apache.tuscany.sca.node.impl.NodeImpl
         start
     [java] INFO: Starting node: Calculator.composite
     [java] 3 + 2=5.0
     [java] 3 - 2=1.0
     [java] 3 * 2=6.0
     [java] 3 / 2=1.5
     [java] 14-Jan-2010 19:34:31 org.apache.tuscany.sca.node.impl.NodeImpl
         stop
     [java] INFO: Stopping node: Calculator.composite

BUILD SUCCESSFUL
Total time: 3 seconds
```

If the calculator sample doesn't work for you, take a look at section A.4, which gives some hints on how to troubleshoot the Tuscany installation. If you still can't get the simple calculator sample to work, you can get help from the Tuscany community by posting the issue that you are seeing on the mailing list. See section A.3 for how to do this.

A.1.4 SCA samples provided with Tuscany

A good way to become familiar with Tuscany is to look at the samples that are bundled with the Tuscany distribution. These are small applications that demonstrate specific SCA and Tuscany features. They're ready to be compiled and run using Ant or Maven. Each sample is accompanied by a README file that explains what the sample does and how to run it. These samples should help you to learn the basics of developing and running SCA applications.

The Tuscany website contains some useful resources that will walk you through getting started with Tuscany. For example, the following link walks you through how to build the Tuscany calculator sample using command-line tools: http://tuscany.apache.org/getting-started-with-tuscany-using-the-command-line.html.

So far we've used Tuscany from the command line. Next we'll look at how Tuscany can be used from an IDE.

A.1.5 Setting up your Java IDE

You can develop SCA applications using your favorite IDE. If you're using Eclipse, you can import Tuscany samples into Eclipse using three alternative approaches: using Maven's Eclipse plug-in, manually without using Maven, and using Eclipse's Maven plug-in. An article at http://tuscany.apache.org/import-existing-tuscany-sca-projects-into-eclipse.html describes these approaches. For developing a new Tuscany application using Eclipse, you can find a step-by-step tutorial at http://tuscany.apache.org/getting-started-with-tuscany.html. A similar approach should also work with other Java IDEs.

A.2 Installing the examples for this book

All of the examples in this book come from the Tuscany SCA Java Travel Sample (hereafter referred to as the travel sample). The code in the travel sample implements the travel-booking application that we're using in this book. The book chapters configure different components of this application in different ways to demonstrate the various features of SCA and Apache Tuscany.

A.2.1 Downloading the travel sample

You can download the travel sample 1.0 (or later) release from the following web page:

http://tuscany.apache.org/sca-java-travel-sample-1x-releases.html

Download the distribution and extract the artifacts from the archive into a directory on your hard drive. The directory path must not contain any spaces. Then you can use Maven to compile all the sample source code by entering the following:

```
cd the_directory_containing_the_sample
mvn
```

Alternatively, you can compile the source code using Ant by entering the following:

```
cd the_directory_containing_the_sample
ant
```

If you're using Ant to compile the source, you'll first need to set the `TUSCANY_HOME` environment variable, as described in section A.2.2.

A.2.2 Setting the TUSCANY_HOME environment variable

The samples in this book require the Tuscany SCA Java runtime. To specify its location for use when running the samples, you need to set the `TUSCANY_HOME` environment variable. For example, on Windows, enter the following:

```
set TUSCANY_HOME=c:\tuscany-sca-1.6
```

On Linux, enter the following:

```
export TUSCANY_HOME=$HOME/tuscany-sca-1.6
```

Now that the travel sample knows where to find the Tuscany runtime, let's try running some sample code.

A.2.3 Testing the travel sample installation

To make sure that the travel sample has been installed and built successfully, let's run a simple example from it. Assuming that you are using Windows, the command would look like the following:

```
cd travelsample\launchers\jumpstart
ant run
```

This example will send a request to the TripProvider component asking for availability of a packaged trip called FS1APR4. If the example runs properly, you should see the following result included in the output on your screen:

```
Trip booking code = 6R98Y
```

If you're not seeing this output, refer to section A.4 for troubleshooting.

A.2.4 Travel sample structure

The main directories in the travel sample are listed in table A.2.

Table A.2 Main directories in the travel sample

Directory	Contents
binaries	Maven scripts for producing a complete set of runtime binary artifacts for the travel sample
clients	Used to show how non-SCA clients can connect to SCA services
contributions	SCA contributions that make up the travel sample
distribution	Maven scripts for building the sample distribution
domainconfig	The domain manager configuration files used in chapters 3 and 12
launchers	Launchers for running the various elements of the travel sample
services	Used to show how SCA references can connect to non-SCA services
testdomain	A test directory where you can run the domain manager and experiment with configuration
util	Utility code and runtime dependencies for the travel sample

The various contributions of the travel sample can be found in the contributions directory. Each chapter in the book identifies which contribution to look at when a sample is referenced.

To run a sample you'll need to use a launcher from the launchers directory. In some cases the launcher name matches the contribution name that it loads and runs. This isn't always true, though—for example, the `fullapp` launcher runs the complete travel-booking application and so loads multiple contributions.

The clients and services directories provide external non-SCA client and service programs to demonstrate various SCA bindings. Typically the client programs call an SCA service, so you'll run a launcher first followed by the client program. The external services are called by an SCA service, so you run the service program first followed by the launcher.

The binaries directory is used to build a self-contained executable set of runtime binary artifacts for the travel sample. This demonstrates how a Tuscany application can be packaged and distributed to end users in binary executable form.

Detailed information about the structure and contents of the travel sample can be found in the travel sample's README file. This file includes a complete list of the launchers and instructions on how to run them.

A.2.5 Using a Java IDE with the travel sample

You can use a Java IDE to run the code in the travel sample. If you're using Eclipse, see http://tuscany.apache.org/import-existing-tuscany-sca-projects-into-eclipse.html for a detailed description of the steps that you need to follow. It's important to import the travel sample code into Eclipse without copying it into the Eclipse workspace, because copying it would change the directory structure and cause problems with some of the code.

A.3 Interacting with the Tuscany project and community

Since its inception, Apache Tuscany has gained a large global community of users and developers with different business requirements and a varying depth of understanding of Tuscany and SCA. Members of this community communicate with one another via the Tuscany mailing lists. Please feel welcome to use the mailing list to ask any questions and get involved in the community.

Tuscany has three main mailing lists, and anyone can subscribe to these lists. Each list has a different purpose, so we'll briefly describe them:

- *User mailing list*—Subscribers to this mailing list are typically Tuscany users and Tuscany software developers. This mailing list is used for discussing usage scenarios and questions related to using Tuscany. Tuscany releases are announced on this list and the developer mailing list.
- *Developer mailing list*—Subscribers are typically Tuscany software developers (core and extension developers). Some users may be interested in watching these discussions because they're related to the ongoing development of the Tuscany runtime and its extensions.
- *Commits mailing list*—Subscribers are typically Tuscany software developers. Code commit notifications and automated test run results are published to this mailing list.

You can subscribe to any of these mailing lists by going to the Tuscany mailing lists page: http://tuscany.apache.org/mailing-lists.html.

To subscribe to a particular mailing list, click the appropriate link. Once subscribed, you'll see email exchanges related to that mailing list, and you can post to the list. You can unsubscribe at any time by going to the same Tuscany mailing lists page and clicking the Unsubscribe link.

- To post to the user mailing list, use the email address user@tuscany.apache.org.
- To post to the developer mailing list, use the email address dev@tuscany.apache.org.

Note that you can send an email to user@tuscany.apache.org without being subscribed to the user list. The disadvantage of this approach is that you won't see all the other users' questions and answers.

You can also watch the activity on the mailing lists by browsing one of the mailing list archives that are listed on the Tuscany mailing lists page. The archive is read-only, and you can't respond to any of the email threads.

Tuscany values user feedback, and the community's goal is to build solutions that solve real IT problems. Please use the mailing lists to share feedback, to contribute, or to suggest new ideas regarding Tuscany. We look forward to talking with you.

A.4 Troubleshooting

If you've tried various options, and you're unable to run an example or your own application, there are several steps that you can take:

1. Check the mailing list archives at http://tuscany.apache.org/mailing-lists.html to see if another user detected the same problem.
2. Check the Tuscany JIRA bug database to see if the problem has already been reported and if a patch is available. Go to http://issues.apache.org/jira/browse/TUSCANY, click the Find Issues tab at the top, and then select Tuscany in the project box on the left panel. Enter your query in the Text Search box on the left panel. Click the View button to get the list of reported problems.
3. If you can't find your answer, post your question on the Tuscany user mailing list including any error messages, information about the Tuscany release you are using, the environment, a pointer to the example or your code sample, and the behavior that you're seeing. Someone from the community will respond to your post.
4. If this turns out to be a bug, you can report it on the Tuscany JIRA bug database at http://issues.apache.org/jira/browse/TUSCANY by creating a new bug.

Please feel free to post the problem on the Tuscany user mailing list to discuss the issue and seek a workaround. When posting on the mailing list, it's good practice to share what you have researched and found thus far.

appendix B
What's next?

You may not be surprised to hear that the Tuscany community has already started implementing the next version of the Tuscany software, building on the 1.x code base, user feedback, and experience gained. We'd like to share the highlights of the future direction of Tuscany 2.x with you, especially if you're interested in participating in the development of this next Tuscany version.

If you have Tuscany 1.x applications that you want to run using the evolving Tuscany 2.x runtime, you need to make a few changes and you need to be aware of a few differences in the runtime. You can see instructions about what to do on the Tuscany website at http://tuscany.apache.org/documentation-2x/converting-tuscany-1x-applications.html.

Tuscany has been a major contributor to the development of the SCA specifications that are now being standardized at OASIS. Therefore, one focus is to move to the standardized version of the SCA specifications. Future releases of Tuscany will continue adding support for new technologies and will integrate with OSGi for improved modularity. It's also becoming clear that Tuscany and SCA offer a service model that's well aligned with the emerging idea of cloud computing. That's another area where we'll see further development and innovation. Let's take a closer look at some of these ideas.

B.1 Support for OASIS Open CSA SCA standards

As part of the standardization process of SCA, OASIS Open CSA is taking the current OSOA SCA version 1.0 specifications as a basis for creating the SCA version 1.1 standards. This work is being done by several OASIS technical committees that are refining and clarifying the SCA 1.0 specifications. Some of the authors of this book are actively involved in the development of SCA standards.

Most of the SCA 1.0 concepts and structures will be retained in the SCA 1.1 standard—implementations, components, composites, services, references, properties,

and so on. But applications will need a degree of modification to be converted from SCA 1.0 to SCA 1.1 because some of the XML schemas and the SCA API Java package names have changed.

The SCA 1.1 standards also include a compliance test suite to ensure that SCA runtimes comply with the standards. Work is currently under way in the Apache Tuscany project to demonstrate compliance with the new SCA 1.1 compliance test suites.

See http://www.oasis-opencsa.org/committees for more information on the OASIS Open CSA SCA 1.1 standards.

B.2 OSGi enablement for the Tuscany runtime

The Tuscany runtime is architected to be modular and extensible. In 1.x, Tuscany uses Maven to achieve this. Maven helps to manage and validate the dependencies between Tuscany modules at build time. Maven's control stays at the JAR level, and it doesn't impose any runtime constraints, so it's difficult to handle versioning of dependencies. This can leave the door open to unnecessary dependencies being introduced, sometimes unknowingly.

Tuscany 2.x adopts OSGi, which helps to formally define Tuscany runtime modules and enforce SPI package dependencies across modules. OSGi also helps Tuscany handle versioning across dependencies. This is a complicated problem because Tuscany integrates with many technologies that can each have dependencies on different versions of the same type of software. Therefore, proper handling of dependency versioning helps ensure predictable behavior.

OSGi also adds dynamicity to the Tuscany runtime so that extensions can be added, updated, or removed without restarting the JVM.

It's worth pointing out that Tuscany continues to work in both OSGi and non-OSGi environments. We don't introduce any hard dependencies on OSGi APIs for the non-OSGi-specific modules.

B.3 Enhanced SCA domain and node support

The Tuscany community continues to enhance Tuscany to ease the development of distributed composite applications. For this reason, there'll be enhancements to the various domain features.

The idea is to allow multiple nodes to be started and automatically form a domain without the need for a separate domain manager. This doesn't mean that the idea of a domain manager won't be supported, but we're trying to make the runtime easier to use in the case where you just want to start a few nodes and have them cooperate without further configuration.

For example, imagine you have one node running in a web application, another node running from the command line, and yet another node running as part of some other software container such as an application server. All of these nodes will be able to exchange metadata about the services they're providing within the domain, and so cross-node domain wires can be created without the need for a central manager.

B.4 Implementation of OSGi remote services with SCA

Outside the world of SCA, OSGi is being enhanced to provide an Enterprise Java programming model. The basic OSGi framework decouples service providers and consumers via a *local* service registry, where a service is an object that one bundle registers and another bundle retrieves. With the introduction of remote services in OSGi release 4.2, OSGi services running on different framework instances can now talk to each other.

With its declarative binding and policy capabilities, SCA fits very well with OSGi remote services. The new version of Tuscany introduces `implementation.osgi`, which can encapsulate one or more OSGi bundles.

The overall architecture is shown figure B.1. The references of an SCA OSGi component represent the OSGi services to be consumed by the bundles. The services of an SCA OSGi component represent the OSGi services provided by the bundles.

Following this approach, `implementation.osgi` can provide the metadata and infrastructure to enable the distribution of OSGi services, and Tuscany enables OSGi bundles to participate in the SCA assembly.

Tuscany also supports the OSGi-centric view. It implements the OSGi Remote Service Admin and SCA Configuration Type specifications (to be published as part of the

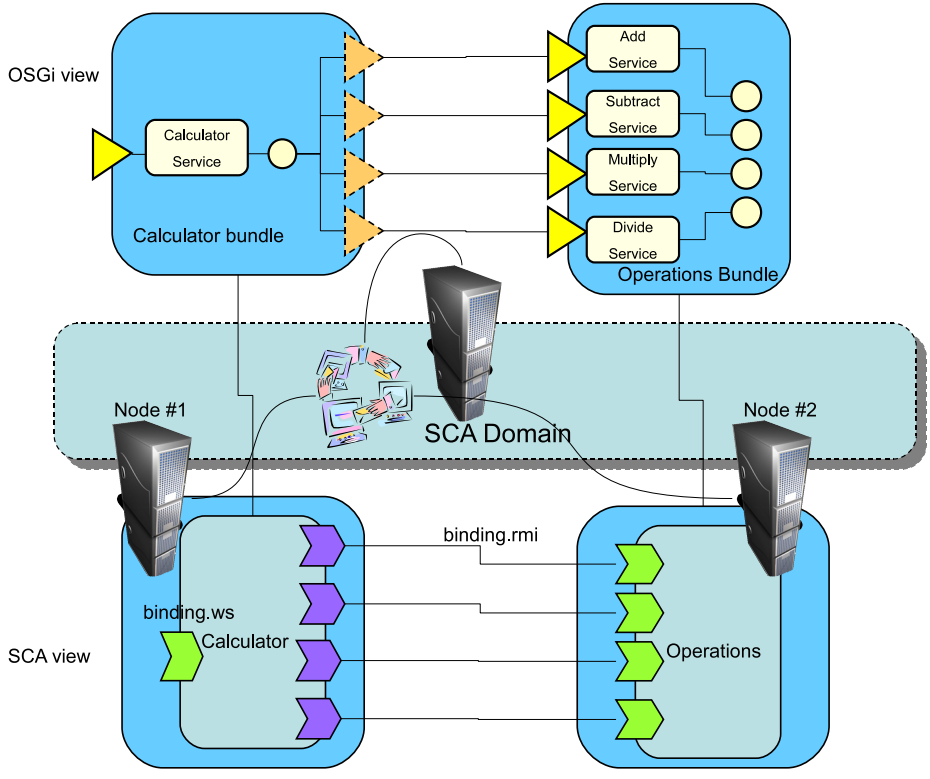


Figure B.1 An SCA domain can be used to provide the inter-bundle wiring of a distributed OSGi application.

OSGi R4.2 Enterprise release). With these capabilities, OSGi Enterprise applications can start to leverage the declarative protocols and polices from Tuscany SCA. Tuscany is the reference implementation chosen by the OSGi alliance for the SCA Configuration Type specs.

B.5 Running Tuscany in the cloud

Cloud computing is an internet-based development model where users rely on the cloud platform, or cloud software provider, to run their business applications. Cloud computing typically means the provisioning of dynamically scalable and often virtualized resources as a service over the internet. Tuscany and SCA's loosely coupled service architecture makes it a good programming model for developing applications in the cloud.

SCA separates all the infrastructure concerns from business logic and makes infrastructure choices declarative. Such declarations make it possible to replace standard infrastructure with cloud infrastructure without changing the business logic.

The cloud is a naturally distributed environment. SCA describes how services can be composed or assembled to create composite applications. Such assembly describes the various natural groups of functions in the application. The cloud infrastructure

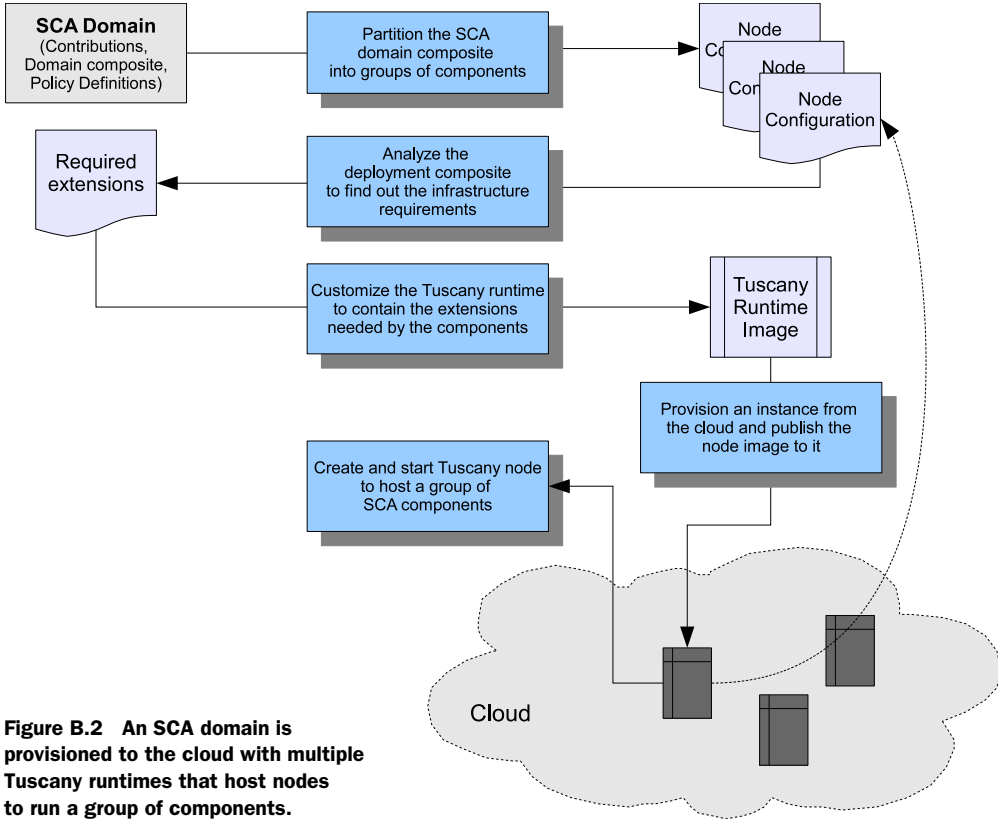


Figure B.2 An SCA domain is provisioned to the cloud with multiple Tuscany runtimes that host nodes to run a group of components.

can take advantage of this information and provision separate groups of functions to different parts of the cloud.

Pulling this together, we can use SCA to remove infrastructure concerns from the application itself. The SCA assembly model can hide the details of the particular cloud infrastructure that's chosen. SCA also provides the opportunity to unify any infrastructure services as reusable SCA components. In this way the differences between different cloud providers can potentially be hidden.

Figure B.2 illustrates some ideas about how the Tuscany domain manager can be enhanced to support provisioning of an SCA domain in the cloud.

The SCA domain can be partitioned into one more nodes that together run a group of components. The infrastructure requirements can be calculated from the composite, and a customized Tuscany runtime can be composed based on the technology extensions needed. The nodes can be provisioned to the cloud based on the infrastructure requirements calculated from the composites.

We're making good progress in all these areas in the Tuscany 2.x code base. At the time of writing, four milestone releases have been made available. You can find the latest code at http://svn.apache.org/repos/asf/tuscany/sca-java-2.x/. Please come and join us to further improve and advance the project: using the code, reporting issues, sharing your requirements, bringing in ideas, fixing bugs, implementing features, or improving documents. You can help, contribute, and be part of the next version of Tuscany SCA.

appendix C OSOA SCA specification license

This book describes the Apache Tuscany 1.x codebase and, as such, describes many of the APIs and annotations that are specified in the OSOA SCA specifications that can be found here (http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications). This information is reproduced under the following license.

The Service Component Architecture Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

> Permission to copy, display and distribute the Service Component Architecture Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Component Architecture Specification, or portions thereof, that you make:
>
> 1. A link or URL to the Service Component Architecture Specification at this location:
>
>   http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications
>
> 2. The full text of the copyright notice as shown in the Service Component Architecture Specification.
>
> BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress Software, Red Hat, Rogue Wave, SAP, Siemens, Software AG., Sun, Sybase, TIBCO (collectively, the "Authors") agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Component Architecture Specification.

THE Service Component Architecture SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE Service Component Architecture SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Service Component Architecture Specification or its contents without specific, written prior permission. Title to copyright in the Service Component Architecture Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

appendix D
Travel sample license

All of the sample code that appears in this book is available as part of the Apache Tuscany project's travel sample. It is reproduced under the following license and can be found here (http://svn.apache.org/repos/asf/tuscany/sca-java-1.x/trunk/tutorials/travelsample/).

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to

compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation

against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks

   This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty

   Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability

   In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability

   While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

index

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X

ENTERPRISE JAVA

Tuscany SCA IN ACTION

Laws • Combellack • Feng • Mahbod • Nash

Tuscany SCA is a technology-neutral infrastructure for building composite applications based on the Service Component Architecture standard. It manages the protocols and other application plumbing, enabling you to focus on business logic and the relationship between services. The resulting applications are more flexible, scalable, and maintainable.

Tuscany SCA in Action is a comprehensive, hands-on guide for developing technology-agnostic, extensible applications. By following a travel-booking example throughout the book, you'll learn how to model, compose, deploy, and manage applications using SCA. The book emphasizes practical concerns, like effectively using Tuscany's supported bindings and protocols and integrating with standard technologies like Spring and JMS to save development time and cost.

What's Inside

- Introduction to Tuscany
- Coverage of Service Component Architecture
- Practical examples and techniques
- Written by core Tuscany committers

This book is for developers interested in service-oriented applications. No experience with Tuscany or SCA is required.

Simon Laws is a member of the IBM Open Source SOA project focused on building the Java runtime for Service Component Architecture (SCA). Coauthors **Mark Combellack**, **Raymond Feng**, **Haleh Mahbod**, and **Simon Nash** are all Tuscany committers.

For online access to the authors and a free ebook for owners of this book, go to manning.com/TuscanySCAinAction

"A great resource."
—Jeff Davis
Author of *Open Source SOA*

"A must-have guide."
—Alberto Lagna, biznology.it

"The A-Z of SCA from the source."
—Ara Ebrahimi
CityGrid Media

"Offers insights and techniques that help you put it all together."
—Doug Warren
Java Web Services

"Start being productive with Tuscany SCA right away."
—Jeff Anderson, Deloitte

ISBN-13: 978-1-933988-89-4
ISBN-10: 1-933988-89-4
55999
9 781933 988894

MANNING $59.99 / Can $68.99 [INCLUDING eBOOK]