Learn by doing: less theory, more results

Railo 3

Easily develop and deploy complex applications online using the powerful Railo Server

Beginner's Guide

Mark Drew
Paul Klinkenberg

Gert Franz
Jordan Michaels

[PACKT] open source*
PUBLISHING
community experience distilled

Railo 3

Beginner's Guide

Easily develop and deploy complex applications online using the powerful Railo Server

Mark Drew

Gert Franz

Paul Klinkenberg

Jordan Michaels

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Railo 3
Beginner's Guide

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2011

Production Reference: 1091211

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-340-1

www.packtpub.com

Cover Image by Rakesh Shejwal (shejwal.rakesh@gmail.com)

Credits

Authors
Mark Drew
Gert Franz
Paul Klinkenberg
Jordan Michaels

Reviewers
A J Mercer
Akbarsait Noormohamed
Jamie Krug
Paul Klinkenberg

Acquisition Editor
Sarah Cullington

Development Editor
Meeta Rajani

Technical Editors
Mohd. Sahil
Lubna Shaikh

Project Coordinator
Joel Goveya

Proofreader
Sol Agramont

Copy Editor
Leonard D'Silva

Indexers
Hemangini Bari
Monica Ajmera

Graphics
Manu Joseph

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Authors

Mark Drew has been developing web applications for a number of clients since the mid 90s. He has been using ColdFusion and writing in CFML since 1996, and even though he has had the occasional forays into Perl, ASP, PHP, Rails, and Java, he is still loving every line of code he has written in CFML.

Mark has been part of the CFEclipse Project developing a CFML IDE and Project Manager for the Reactor ORM Project, as well as contributor to a number of frameworks.

His career has concentrated on e-commerce, web content management, and application scalability for various well-known brands in the UK as well as the rest of the world.

Mark is also a well-known speaker at various conferences on subjects close to his heart, such as ORMs, Frameworks, Development Tooling and Process, as well as noSQL databases and a range of other topics.

Mark lives in Greenwich, London, where the Mean Time comes from. Mark isn't mean of course. He works as the CEO of Railo Technologies Limited (`http://www.getrailo.com`), a web development consultancy and Professional Open Source provider of Railo Server.

Really, this book would not have been possible without the help of the below-mentioned people, whom I am utterly indebted to and I shall fulfill all those promises to buy them a beer, even if it takes emptying out a whole brewery. I want to thank Gert Franz, for giving me the opportunity to write this book; Sarah Cullington, for her invaluable advice as an editor; Joel Goveya, for his reminders and motivation to get all the chapters done on time; Paul Klinkenberg, for his hard work and timely offers of help; Roland Ringgenberg, for his Flex and Flash mastery—I would have really been out of my depth on that one! I would also like to thank Sean Corfield, Peter Bell, and A J Mercer, for their awesome feedback on chapters in the process of writing this book; Todd Rafferty, for his great contributions and eagle eye; Andrea Campologhi, for his stellar AJAX skills and contributions to Railo Server; and Michael Offner, for all his skills in developing Railo Server itself and giving me peeks behind the curtains to how it all works. A big thank you to all the folks in the Railo Users mailing list for keeping the community alive and kicking. Finally, I would like to thank Charlie Khan and The Organ Grinder for the musical accompaniment that helped clarify my thoughts as I went along!

Gert Franz was born in 1967 in Romania. He moved to Germany in 1982. He studied Astrophysics in Munich in the early nineties and lives in Switzerland since 1998.

Gert is a father of three children and lives in with his Swiss girlfriend, somewhere next to Zurich. Even though the jobs Gert had did not involve Astronomy in any way, he still remained loyal to it as a hobby and from time to time he taught local classes about the wonders and miracles of Astronomy.

In the past 20 years, he worked as a Senior Programmer for several different companies and leads Railo Technologies Switzerland as a CEO since its foundation in 2007.

Gert is a well-known speaker who appeared and appears at several different conferences around the world. Mostly, he speaks about Railo and/or performance tuning. Besides speaking, Gert programs a lot, and does all different kinds of consulting related to Railo, CFML, databases, and system architectures. He is a specialist in performance tuning, especially with MSSQL and Railo.

Next to the things mentioned before, Gert hosts Railo training sessions and performance-tuning training sessions around the world. Along the way, Gert acquired a deep knowledge in Railo, CFML, Delphi, C, ASP, SQL, SQL tuning, and other programming-related things.

Paul Klinkenberg (1979) is a long-time CFML addict, living in The Netherlands with his wife Emma and baby daughter Luce. His history in both Commercial Economics and Fine Arts were no match for the enthusiasm he got from programming. In his 10+ years of experience in programming in CFML, he has always been investigating and pushing the boundaries of this magnificent language constantly. As a Railo Team member, he is in charge of managing and promoting Railo Extensions. He never stops thinking and creating new features for Railo Server, and tries to evangelize Railo as much as possible.

He shares code projects and ideas via his weblog `http://www.railodeveloper.com`. Though it has gotten a lot quieter on his blog lately, as his beautiful baby daughter Luce, born in 2011, gets a lot of his attention.

Paul is currently employed at the Dutch web-development company and Railo partner Carlos Gallupa BV. He is also working on projects through his own company Ongevraagd Advies, which means *unasked advice*. Friends and clients often say the name suits him really well, with his power to thoroughly analyze project plans and ideas, and come up with new ideas and suggestions out of the blue.

I'd like to sincerely thank my lovely and caring wife for the patience she had with me. It's probably not easy to share your husband with a programming language. To Luce: je papa houdt van jou, schatje!

Jordan Michaels currently participates in the Railo Team as the Community Deployments Coordinator, where his duties include coordinating efforts and documentation on how to deploy Railo in various environments. Jordan has been a CFML enthusiast and developer for just over 8 years, and is now the co-owner of Vivio Technologies where he operates as a CEO. Jordan is an active participant in the CFML community providing evangelism, community support, and has also printed various articles on CFML. Jordan is also an amateur musician and science buff. Jordan currently resides with his wife and two sons in WA state, USA.

About the Reviewers

A J Mercer first discovered CFML as a DBA when looking for a way to extract data from Informix and display it with links to drill down to detailed information. That was back in 1997 when that was a big deal. After battling with CGI scripts and embedded ESQL in C and Informix 4GL he discovered Cold Fusion Express. This is exactly what he was looking for, and with the added bonus of being able to email reports – via a scheduled task!

After a job or two doing all sorts of consultancy development work in various web and desktop languages, he was approached by a firm and asked if he knew anything about ColdFusion. This was in 2000 when being able to spell CFML was enough to get you hired. It was in this job that he developed his web development skills using ASP and CFML. Luckily for him, the development team was big enough to allow for specialization and was allowed to just work on the CFML projects. During web development team meetings his favorite joke when the .NET guys were stuck on something was "Allaire / Macromedia have got a patch for that—it is called ColdFusion". It was also at this job when he first discovered FuseBox and introduced a development standard into the organization.

AJ has backed his career on CFML and has swapped jobs when the pointy-haired bosses started phasing out ColdFusion. He is deeply passionate about CFML and has been actively promoting the product and sharing his knowledge with local user group CFUGWA (of which he was manager for 5 years) and has presented at webDU and cf.Objective(ANZ).

He is one of many who subscribe to the theory that CFML needs a free version to be able to compete with the likes of .NET, PHP, and Ruby. In his spare time, he was on the look out for other CFML engines. In 2006, he discovered Railo—and once again stopped looking. He worked with many Framework developers, such as Farcry CMS, MangoBlog, ColdBox and Mach-II, and the Railo team to get these frameworks running on Railo. Due to his passion and enthusiasm, he was appointed Railo Community Manager for Australia in 2010.

> I feel humbled and honored to have been asked to review this book. The Railo team is made up of a lot of people I respect and look up to in the CFML community. My hat goes off to Mark Drew for taking on this mammoth task of writing this book. Truth be told, there was not a lot I had to do as a reviewer, and I learned quite a few things on the way through at the same time, as I am sure you will too.
>
> I will also take this opportunity to thank and congratulate Michael Offner and Gert Franz for Railo—not just the Server product, but the Team and Consultancy. Way back, when I first started out with Railo, Gert was very generous with his time and helped me build my Railo server. Gert and the rest of the team still, to this day, are passionate about helping people with Railo and CFML. So, this book is not the end of your learning, but just the start of the exciting world of Railo. Enjoy!

Akbarsait Noormohamed is a passionate Computer programmer and has been a ColdFusion developer since 2004. Akbarsait specializes in using CFML, SQL (MS SQL Server, MySQL, and Oracle), and web technologies for creating web applications and Content Management Systems.

Akbarsait is currently working as a Consultant for MindTree Ltd in Chennai India. His experience includes building web applications and intranet systems for Travel and Transportation, Healthcare, and ERP domains. He loves troubleshooting and solving problems in CFML engines. He has always had a keen interest in improving web performance.

He also manages the Chennai's ColdFusion User Group in India and he is an Adobe Community Champion for ColdFusion. He currently holds a B.E in Computer Science and Engineering and Diploma in Electrical and Electronics Engineering from Bharathidasan University. You can follow him on his blog at `http://www.akbarsait.com` or at `@Akbarsait` on Twitter.

Jamie Krug developed a love for programming early on, writing a BASIC program on a RadioShack TRS-80 to track "little league" baseball batting averages at an early age. He has since then continued to enjoy programming and the learning experiences along the way. Primarily building web applications in CFML since 2001, Jamie is a passionate learner and also geeks around in Java/Groovy, Flex/ActionScript and Linux, among others. He also greatly appreciates and participates in many open source software projects. You'll find Jamie occasionally blogging at `http://jamiekrug.com/blog/`.

I'd like to thank my lovely wife, Wendy, and children, Ayvin and Nyah, for their loving support. I'd also like to thank my entire network of friends and family everywhere. I "work" doing something I truly enjoy, and enjoy constant support and encouragement all through. For my experience and learning opportunities, I thank the amazing CFML community, as well as the countless passionate software geeks everywhere.

Table of Contents

Preface

Railo Server is one of the quickest ways to start developing complex web applications. Widely considered as the fastest CFML (ColdFusion Markup Language) engine, Railo Server allows you to create dynamic web pages that can change depending on the user input, database lookups, or even the time of day.

Railo 3 Beginner's Guide will show you how to get up and running with Railo Server, as well as enabling you to develop your web applications with ease. You will learn how to install Railo Server and the basics of CFML as the book progresses to allow you to gradually build up your knowledge and your dynamic web applications.

Using Packt's Beginner's Guide approach, this book will guide you with step-by-step instructions, through installing the Railo Server on various environments. You will learn how to use caches, resources, event gateways, and special scripting functions that will allow you to create web pages with limitless functionality. You will even explore methods of extending Railo by adding your own tags to the server and building custom extensions. Railo 3 Beginner's Guide is a must for anyone getting to grips with Railo Server.

What this book covers

Chapter 1, *Introducing Railo Server*, gives an introduction to Railo Server and also shows us an overview of how it is a breeze to develop web applications.

Chapter 2, *Installing Railo Server*, describes how to install Railo Server under a number of operating systems as well as using different servlet containers.

Chapter 3, *CFML Language*, provides a foundation for using the CFML Language to develop sites in Railo Server. This chapter also covers object-oriented programming with components as well as functions and tags.

Chapter 4, *Railo Server Administration*, details the functionality in the server and web context. It also explains how different settings affect the behavior of the server and cover a number of other topics, such as Extension , Archives and Resources, and Security.

Chapter 5, *Developing Applications with Railo Server*, looks at how applications can be defined programmatically, the `Application.cfc` lifecycle, and also how components interact with the database and even looks into various caching techniques.

Chapter 6, *Advanced CFML Functionality*, looks at the scripting formats available in CFML, while investigating the CFScript language. It also looks at the built-in components available in Railo Server.

Chapter 7, *Multimedia and AJAX*, this practical chapter goes through converting and displaying video, as well as communicating between the browser and the server using the AJAX functionality of Railo Server.

Chapter 8, *Resources and Mappings*, describes how to use local and remote resources via the use of mappings within Railo Server. It also looks at how we can use ZIP and TAR files, using RAM as a handy resource and saving our files out in the Cloud using Amazon S3.

Chapter 9, *Extending Railo Server*, looks at how we can create new tags and functions for Railo Server and create an extension so that we can share our changes to the core server with other Railo Server users via our own Extension Provider.

Chapter 10, *Creating a Video-sharing Application*, brings together all your skills into a single application, setting up the Object Relational Model (ORM), creating security, converting your videos, and displaying your videos for everyone to use!

What you need for this book

You can run Railo server on a PC or a Mac, under Windows, OS X, and Linux.

To edit the code snippets described in the book, you will need a text editor, such as TextMate on OS X or Textpad on Windows. As long as you are able to edit text files the choice of software is up to you.

Who this book is for

If you want to develop your own dynamic web applications using CFML, then this book is for you. No prior experience with Railo or CFML is required, although you are expected to have some experience in web application development and the knowledge of HTML, basically, how websites work in general.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: " Using the `<cfvideo>` tag, we are able to convert to a number of formats."

A block of code is set as follows:

```
<script type="text/javascript" charset="utf-8">
    onError = function(code,message){
      alert(code + ' - ' + message);
    }
    displayTodos =   function (data){
      document.getElementById('taskname').value = “";
      Railo.Ajax.refresh('displayTodos');
    }
</script>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<head>
  <link rel="stylesheet" href="main.css" type="text/css">
  <title>Todo</title>
  <cfajaxproxy bind="cfc:todo.TaskService.addTodo({taskname})"
         onSuccess="displayTodos"
         onError="onError"/>
<cfajaxproxy cfc="todo.TaskService"" jsclassname="TaskService">
```

Any command-line input or output is written as follows:

```
sudo vi /etc/default/jetty
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: “Go to **Members ¦ Login** and use your new username and password to log in to the website."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.PacktPub.com. If you purchased this book elsewhere, you can visit http://www.PacktPub.com/support and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/support, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introducing Railo Server

The Web is now the best way to deploy your applications. It is because of the ease of use and reach to your users and, of course, the fact that you only have to deploy one version of your code for everyone to use.

To get this done, you would have probably looked at different languages and even frameworks out there. Did they seem complicated to get going? Were there lots of new terms to learn?

This is where Railo Server comes in. It provides an easy way to write and deploy your applications using a language that is very similar to HTML (`http://en.wikipedia.org/wiki/HTML`), *about which you'd already know if you happened to work with static websites.*

If you are already using HTML, Railo Server is a perfect addition to your toolkit!

In this book, you will learn how to use Railo Server to develop web applications in a very efficient manner. We will also introduce the different features and options available to get things done easily and quickly as we go along.

In this chapter, we will:

- Introduce you to the Railo Server
- Introduce you to some of the features of Railo Server
- Show you why Railo Server and CFML make developing web applications a breeze

Let's dive right in!

Why use Railo Server?

HTML per-se (without JavaScript) is a static language, which means that you cannot query databases, send e-mails, execute searches, or generally interact with the server, and return dynamic results to the user.

Imagine that you need to do one of the above tasks. HTML doesn't offer any solutions for these kinds of problems because it is just a way to display information and not manipulate other systems.

What can we do in order to overcome this problem?

ColdFusion Markup Language (CFML) neatly fits into the HTML syntax and allows you to place CFML tags in-between HTML tags. Basically, Railo Server generates HTML, which is then interpreted by the browser.

Depending on the tags that you are using, different HTML pages are sent back to the client.

Let's have a look at an example Railo Server template:

```
<html>
<head><title>An Example Template</title></head>
<body>
<div id="AccountHeader">
  <cfif SESSION.loggedIn>
    <h3>Private bank account No.:
    <cfoutput>#bank.accountNr#</cfoutput>
    Show secret information
  </cfif>
</div>
</body>
</html>
```

The highlighted code above shows you some CFML (or Railo Template) code. This code is dynamic and will run on the server before returning the rendered HTML back to the browser that requested the page.

If you are used to reading HTML code, you can easily figure out what the previous code is supposed to do. (It displays the HTML between the `<cfif>` tags if the variable `loggedIn` in the `SESSION` is true.)

What does Railo Server do?

Railo Server is a service that can be installed on any Java Application Server (`http://en.wikipedia.org/wiki/Java_Servlet`) that helps you write web applications quickly and easily, without the complexities normally associated with developing fast, scalable, and secure applications in Java.

Railo Server is a servlet that runs on any servlet engine. Servlets are small (or large) programs that are invoked by a servlet engine or a J2EE application server (such as Tomcat, JRun, Jetty, Resin, Glassfish, IBM WebSphere, BEA WebLogic, and others).

The application servers run within the Java Runtime Environment (JRE) and call the corresponding servlet; if a certain request comes to it, it matches certain conditions.

If this is the first time you are hearing about servlets, JREs, and similar things, no need to worry, Railo Server can be installed with an easy installer for a complete setup. It installs the JRE, the servlet engine, and can even hook into an existing IIS or Apache web server. What the servlet does is completely open. In this case, the Railo Servlet allows us to do the following:

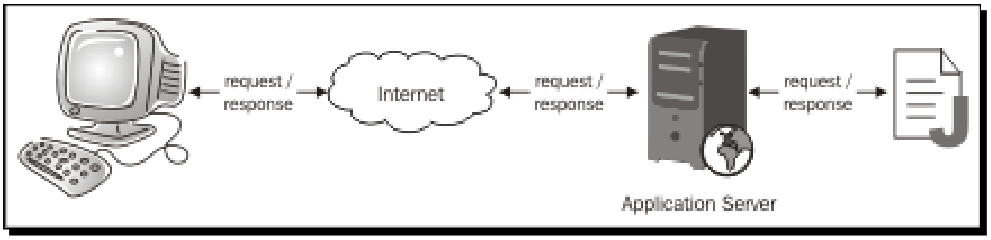
- Compile CFM files into Java bytecode
- Check the syntax of invoked files
- Invoke the necessary files according to the CFML syntax
- Execute the bytecode and throw any errors that may occur
- Build up a request environment
- Interact with external resources, including:
    - Databases
    - Cache systems
    - Filesystems (virtual or physical)
    - Web services
    - Event Gateways
    - Write files
- Return HTML, JSON, XML, PDF, or anything else that an HTTP request can generate and handle
- Execute scheduled tasks
- Send e-mails
- Create and read RSS and ATOM feeds
- Manage sessions

If you need to build a dynamic web application, Railo Server comes into play. You can use Railo Server in order to program your applications in CFML and run your application either in an externally-hosted server or a local server.

Basically, when a client enters a URL into his/her browser address line a request is made to a Railo Server. The process is the following:

1. The URL is resolved to an IP address by checking the DNS system.
2. The browser sends the request to the server at the retrieved IP address.
3. The web server on the target server is invoked (since that runs on port 80).
4. The web server checks what kind of extension or filter definition matches the URL (`.cfm` or `.cfc`).
5. The application server is invoked (assuming it is responsible for the extension `.cfm`).
6. The application server calls the corresponding servlet (in our case, Railo).
7. Railo processes the request.
8. Railo sends back HTML (or whatever else) to the application server.
9. The application server sends the response back to the web server.
10. The webserver sends the HTML response back to the client.
11. The client browser interprets the returned HTML.

So, basically with Railo, you will dynamically create static pages. The following image illustrates this process:



In essence, Railo Server provides the web developer with CFML, which is a simple yet powerful language, as well as a highly configurable server. This helps you write applications extremely quickly.

Railo itself is written in Java, and therefore the libraries deployed with Railo are JAR files.

Your CFML code will be compiled to Java, so you could say that programming in Railo CFML is ultimately programming in Java because Railo generates Java and disguises the complexity of Java applications from the programmer.

A better look at Railo Server

Whenever you use a programming language, the features and tools that come with the language are most influential in helping you decide whether you made the right choice for the current project.

Railo Server's strengths are geared towards **Rapid Application Development** (**RAD**). Once you use Railo Server, you will find that you can create websites literally within hours. For example, you can build a blog within an hour or two.

Time is a huge asset and any tool or any programming language that saves you time is something you should take a closer look at.

Let's take a look at how easy it is to send an e-mail with Railo in comparison to other programming languages:

First of all, this is how you would send an e-mail in PHP:

```
<?php
   require_once "Mail.php";
   $from = "Sandra Sender <sender@example.com>";
   $to = "Ramona Recipient <recipient@example.com>";
   $subject = "Hi!";
   $body = "Hi,\n\nHow are you?";
   $host = "mail.example.com";
   $username = "smtp_username";
   $password = "smtp_password";
   $headers = array ('From' => $from,
         'To' => $to,
        'Subject' => $subject);
   $smtp = Mail::factory('smtp', array ('host' => $host,
              'auth' => true,
              'username' => $username,
              'password' => $password));

   $mail = $smtp->send($to, $headers, $body);

   if (PEAR::isError($mail)) {
     echo("<p>" . $mail->getMessage() . "</p>");
    } else {
     echo("<p>Message successfully sent!</p>");
    }
     ?>
```

Let's have a look at the same functionality in Java:

```
      import javax.mail.*;
      import javax.mail.internet.*;
      import java.util.*;

      public void postMail( String recipients[ ], String subject, String
    message , String from) throws MessagingException
      {
          boolean debug = false;
           //Set the host smtp address
           Properties props = new Properties();
           props.put("mail.smtp.host", "smtp.jcom.net");

          // create some properties and get the default Session
          Session session = Session.getDefaultInstance(props, null);
          session.setDebug(debug);

          // create a message
          Message msg = new MimeMessage(session);

          // set the from and to address
          InternetAddress addressFrom = new InternetAddress(from);
          msg.setFrom(addressFrom);

          InternetAddress[] addressTo = new InternetAddress[recipients.
    length];
          for (int i = 0; i < recipients.length; i++)
          {
              addressTo[i] = new InternetAddress(recipients[i]);
          }
          msg.setRecipients(Message.RecipientType.TO, addressTo);

          // Optional : You can also set your custom headers in the Email
    if you Want
          msg.addHeader("MyHeaderName", "myHeaderValue");
          // Setting the Subject and Content Type
          msg.setSubject(subject);
          msg.setContent(message, "text/plain");
          Transport.send(msg);

      }
```

And finally, let's compare that to how we would do it in CFML:

```
        <cfmail from="Sandra Sender <sender@example.com>"
          to="Ramona Recipient <recipient@example.com>"
          subject="Hi!">
      Hi,
      How are you?
        </cfmail>
```

As you see, this is a lot of functionality with very little code!

Another comparative example would be the number of lines of code required to create a web service. This shows you how Railo Server really can speed up your development:

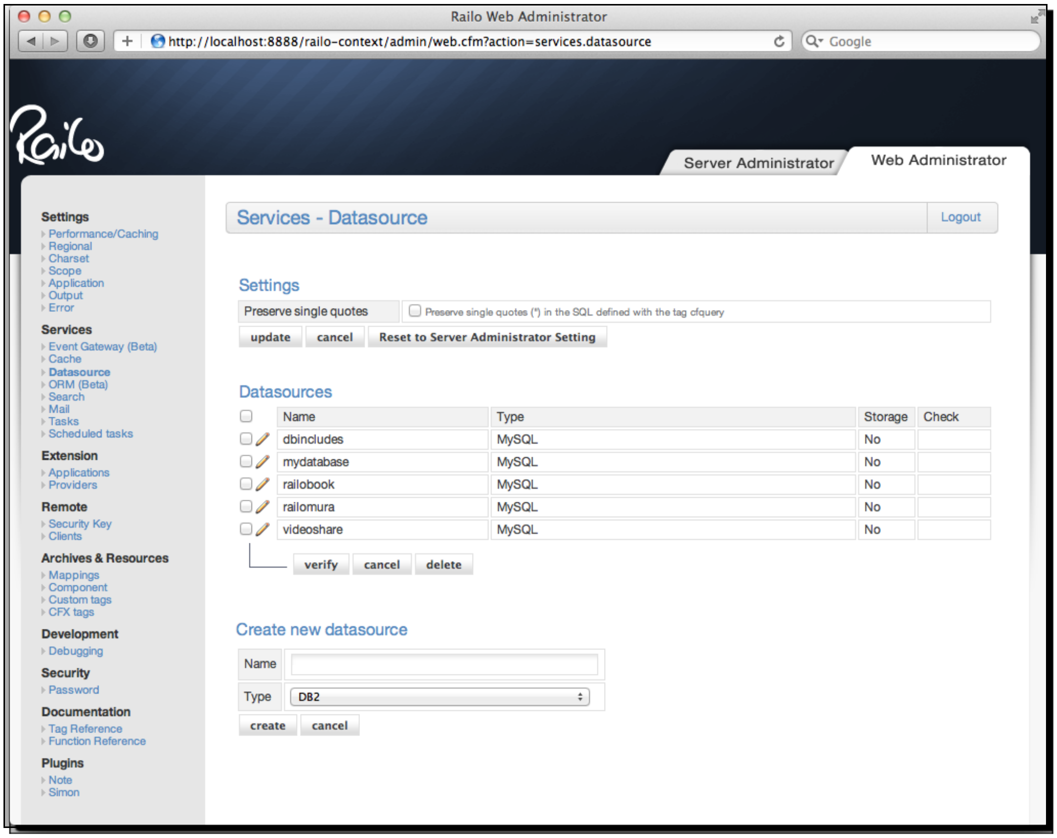
PHP:25 lines

Java:16 lines

Railo:1 line

In terms of productivity, Railo Server is very powerful and can match any other programming language in performance and especially in conciseness, as we will discover with many of the examples in this book.

In addition, Railo Server lets you manage your resources such as databases, mail servers, and even the behavior of the server itself with the included Railo Administration application.



Services - Datasource

Settings

Preserve single quotes □ Preserve single quotes (") in the SQL defined with the tag cfquery

Datasources

□	Name	Type	Storage	Check
□	dbincludes	MySQL	No	
□	mydatabase	MySQL	No	
□	railobook	MySQL	No	
□	railomura	MySQL	No	
□	videoshare	MySQL	No	

Create new datasource

Name:
Type: DB2

Another feature that the Railo Server includes is session management, allowing the state of users to be persisted across several requests.

This feature is also expanded in being able to store general code settings in an application scope. This is kept in memory so that the environment doesn't have to be built up on each request; this, means great performance benefits right out of the box for your application.

With Railo Server, scaling and clustering is extremely easy to achieve without having to configure a full-blown J2EE cluster.

Starting with basic CFML, we can explore the full power of Railo components, caching features, database interaction, and last but not least, Java interaction.

What else can you do with Railo Server?

This section will outline some of the features that also come with Railo Server and will give you some idea of the power behind it.

CFML compiler

Railo Server has an integrated bytecode generator (written with ASM) that directly compiles CFML written templates into Java bytecode. The compiler is very fast and, on average, it compiles a template file in less than 10 milliseconds.

In early versions, Railo Server translated CFML code into XML, then used XSL to convert it into Java, and finally used a Java compiler to generate bytecode. Since version 2.0, Railo has had an integrated bytecode generator. The compiler is able to spot syntax errors in templates and throw errors accordingly.

Railo archives—compiled code

Railo Server is capable of using Railo archives that are compiled versions of your templates. This allows you to package and distribute your applications without having to worry about other people viewing your code and possibly stealing your intellectual property. In addition, the size of your application decreases drastically and it executes a lot faster (as it is already compiled).

Wide variety of CFML tags and functions

Railo Server has more than 125 tags and over 500 functions that allow you to create the best applications in no time. There are tags and functions for nearly every task you can think of, from manipulating arrays to converting images. The great thing is that Railo Server is also extendable, so you can create your own functions and tags to extend your application or the server itself.

Object-oriented approach

CFML components give you the power you need in order to scale and design MVC-based applications. They allow you to use Object Oriented Programming (OOP) techniques such as methods, encapsulation, and inheritance. This leads to robust, encapsulated code, and, in the long run, reduced maintenance costs.

Scripting support

Railo Server also integrates a scripting version of the tag-based programming language. Programmers who are familiar with the coding syntax of other languages, such as JavaScript, PHP, or JSP, will enjoy this feature.

For example, you can create a component using tags as follows:

```
<cfcomponent output="false">
  <cffunction name="init">
    <cfreturn this />
  </cffunction>
</cfcomponent>
```

You can also use the `cfscript` format to achieve the same thing:

```
component output=false{
  function init(){
    return this;
  }
}
```

Integrated administration frontend

With the web and server administrator, Railo Server offers a very easy tool in order to configure the behavior of local and global applications. The Web and Server Administrator applications are the main tools you will use in order to interact with the behavior of Railo. Of course, these applications are also built using CFML. So, you can programmatically adjust all the settings from CFML itself.

Background task execution

Railo Server integrates a task manager that allows you to asynchronously execute requests in the background.

Extension manager

Railo Server tries to include everything you need, but sometimes there are things that are very specific to your application. For this, there is an Extension Manager that allows you to add features to Railo Server directly from the Railo Server Administrator application. The extension store offers programmers a whole new set of features and applications that are easily installed and updated.

Easy update mechanism

Extensions are not the only thing that can be easily updated. You can update Railo Server itself to the latest version with just a click. In the Railo Server administrator, you will get notifications as soon as a new release of Railo Server is available and this allows for a one-click update. If you need to restore the old version again, it is also just one click away. Normally, it is only a matter of seconds.

Compatibility

When developing Railo Server, it was a strict goal to keep compatibility with the CF standard as tightly as possible. This is demonstrated by the fact that with various applications, a change of the application server to Railo did not change anything in the runtime behavior of the application itself, except maybe for the improved speed. So, if you already have some CF applications running, there's no reason to fear high migration costs.

Framework and application compatibility

At the moment, all of the major CFML frameworks or Content Management Systems (CMS) work with Railo Server. So, if you are used to using a framework or tool like FW/1, ColdSpring, ModelGlue, CFWheels, or ColdBox, you don't have to fear incompatibilities. In fact, FW/1 is even written by one of the members of the Railo team, Sean Corfield.

Security

With Railo Server, global security settings can be made for all applications running on the server. For example, access to the filesystem can be denied for a single application or restricted only to files that lie within the web root of the application.

Virtual filesystems

In Railo Server, it is very easy to interact with different virtual filesystems (VFS). The local hard disk is just an instance of a virtual filesystem. Other VFSs that Railo supports are RAM, HTTP, DB, FTP, SFTP, ZIP, TAR, S3, and others.

High performance

Railo Server's main goal was to be the CFML engine with the best performance. One of the main reasons why this goal can be achieved is because Railo uses common resources for all applications. Additional changes in the architecture and various internal structures allowed us to push the performance to higher limits. To the end user, these changes are noticeable in a short response time, during the execution of the same code on the various engines.

Easy installation

The easiest way to give Railo Server a try is to download Railo Express, unpack it, and hit a batch file. There is no easier way to install the software. Railo Server can also be downloaded as an integrated installer for various operating systems and will install Apache Tomcat and add connectors from web servers, such as Microsoft's IIS and the Apache HTTP server.

Inexpensive and free

Railo Server is free and an open source LGPL V2 allowing you to both use and re-distribute your applications with the underlying engine.

When it comes to using Railo Server in a clustered environment, there are some useful and inexpensive extensions that can be purchased. Just use the Railo Extension Manager that is part of the Railo Administration Application in order to see what extensions are available.

Because Railo Server is free, you don't have to fear any update cost or high-initial cost if you plan to use it in a large environment.

Easy clustering

Railo Server makes scaling and clustering very easy. You are able to build independent nodes that act as a virtual cluster and meet any scalability demand your application may have.

Summary

Hopefully, this chapter has given you an overview of what Railo Server offers the web developer in terms of ease of programming, conciseness of language, and feature set.

You should now have an idea of:

- The small number of lines you need to write to get things done
- The rich number of features that Railo Server provides
- How easy it is to extend Railo Server by using extensions
- The way templates are processed and delivered to the client
- The powerful Java underpinnings that are made available to you without any complexity

In the next chapter, we shall have a look at the various ways you can install Railo Server and how to get up and running quickly under different environments.

2
Installing Railo Server

Let's get started with Railo Server! The first thing we want to do is install it on our computer. Luckily, this is pretty easy.

Railo Server is essentially a Java web application that can be installed on many servlet containers such as Jetty, Tomcat, and Resin. In this chapter, we are going to look at the three different ways to install Railo Server

In this chapter, we will:

- Get up and running with Railo Express
- Get up and running with the Railo Server Tomcat installer in Windows environment
- Get up and running with the Railo WAR and Jetty in the Linux environment

So, let's get on with it!

Getting up and running with Railo Express

Running Railo Express is probably the quickest way that you can get started with Railo.

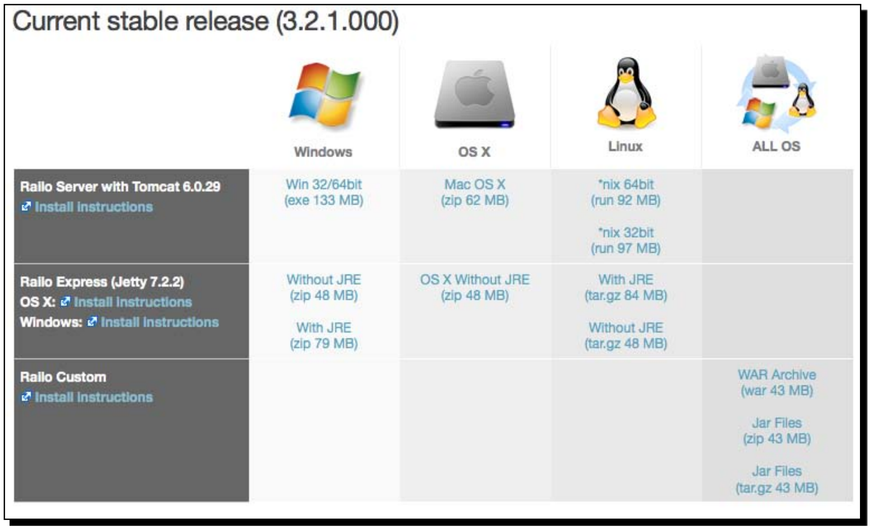
Railo Express includes Jetty, which is a very lightweight servlet container and a great way to get a local development version of Railo running on your machine. This is very helpful when you want to try out the code samples in just a few minutes without having to permanently install any software.

> Servlet containers are a way to run Java applications to be served by a web server. They can be run standalone or connecting to a web server such as Apache or Microsoft's IIS.

Time for action – downloading Railo

The following Railo Express procedure is for OS X, but it is identical to how we do it on Windows. Let's get started!

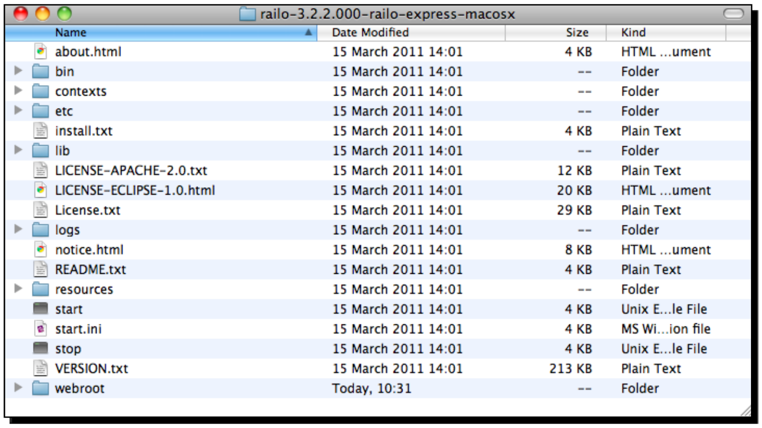
1. To get started, we should first head to `http://www.getrailo.org/index.cfm/download/`.
2. Scroll down to the **Current stable release** table and select the correct version for your operating system (in this case, we are downloading the Railo Express for OS X).



Current stable release (3.2.1.000)

	Windows	OS X	Linux	ALL OS
Railo Server with Tomcat 6.0.29 Install instructions	Win 32/64bit (exe 133 MB)	Mac OS X (zip 62 MB)	*nix 64bit (run 92 MB) *nix 32bit (run 97 MB)	
Railo Express (Jetty 7.2.2) **OS X:** Install instructions **Windows:** Install instructions	Without JRE (zip 48 MB) With JRE (zip 79 MB)	OS X Without JRE (zip 48 MB)	With JRE (tar.gz 84 MB) Without JRE (tar.gz 48 MB)	
Railo Custom Install instructions				WAR Archive (war 43 MB) Jar Files (zip 43 MB) Jar Files (tar.gz 43 MB)

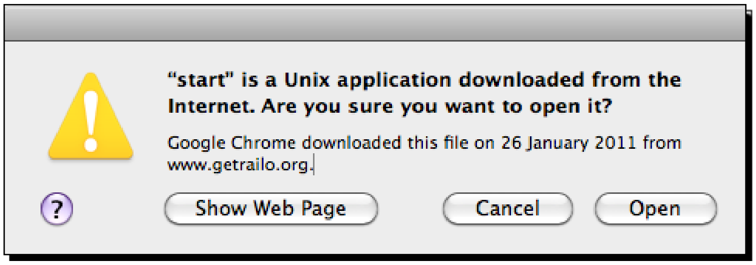
3. Once you have downloaded the ZIP file, extract it and we will have a folder like the one shown in the following screenshot:



railo-3.2.2.000-railo-express-macosx

Name	Date Modified	Size	Kind
about.html	15 March 2011 14:01	4 KB	HTML ...ument
bin	15 March 2011 14:01	--	Folder
contexts	15 March 2011 14:01	--	Folder
etc	15 March 2011 14:01	--	Folder
install.txt	15 March 2011 14:01	4 KB	Plain Text
lib	15 March 2011 14:01	--	Folder
LICENSE-APACHE-2.0.txt	15 March 2011 14:01	12 KB	Plain Text
LICENSE-ECLIPSE-1.0.html	15 March 2011 14:01	20 KB	HTML ...ument
License.txt	15 March 2011 14:01	29 KB	Plain Text
logs	15 March 2011 14:01	--	Folder
notice.html	15 March 2011 14:01	8 KB	HTML ...ument
README.txt	15 March 2011 14:01	4 KB	Plain Text
resources	15 March 2011 14:01	--	Folder
start	15 March 2011 14:01	4 KB	Unix E...le File
start.ini	15 March 2011 14:01	4 KB	MS Wi...ion file
stop	15 March 2011 14:01	4 KB	Unix E...le File
VERSION.txt	15 March 2011 14:01	213 KB	Plain Text
webroot	Today, 10:31	--	Folder

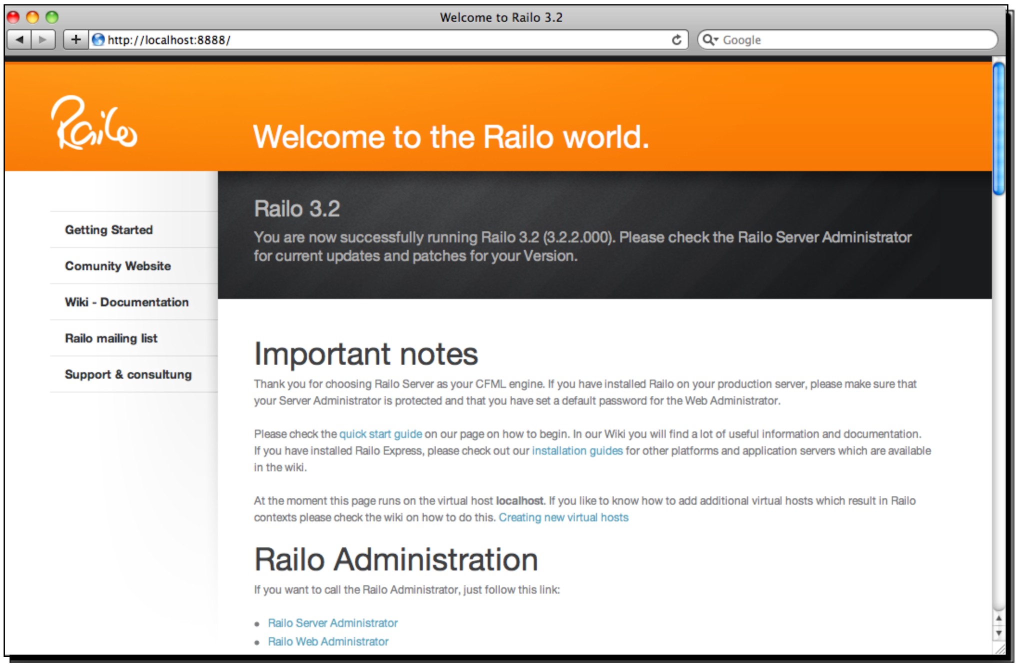
4. Now that we have expanded the ZIP file, you can start Railo Express by clicking on the `start` file (`start.bat` on Windows). If we get a security warning, just click on **Open**.



"start" is a Unix application downloaded from the Internet. Are you sure you want to open it?

Google Chrome downloaded this file on 26 January 2011 from www.getrailo.org.

5. A terminal window will open and start a list of commands; this is Railo Server starting up. Once the commands stop running, you can check that Railo Express is running correctly by going to `http://localhost:8888`, where you will get Railo's welcome screen:



Welcome to the Railo world.

Railo 3.2

You are now successfully running Railo 3.2 (3.2.2.000). Please check the Railo Server Administrator for current updates and patches for your Version.

Important notes

Thank you for choosing Railo Server as your CFML engine. If you have installed Railo on your production server, please make sure that your Server Administrator is protected and that you have set a default password for the Web Administrator.

Please check the quick start guide on our page on how to begin. In our Wiki you will find a lot of useful information and documentation. If you have installed Railo Express, please check out our installation guides for other platforms and application servers which are available in the wiki.

At the moment this page runs on the virtual host **localhost**. If you like to know how to add additional virtual hosts which result in Railo contexts please check the wiki on how to do this. Creating new virtual hosts

Railo Administration

If you want to call the Railo Administrator, just follow this link:

- Railo Server Administrator
- Railo Web Administrator

6. Hurray! Railo Server is running!

What just happened?

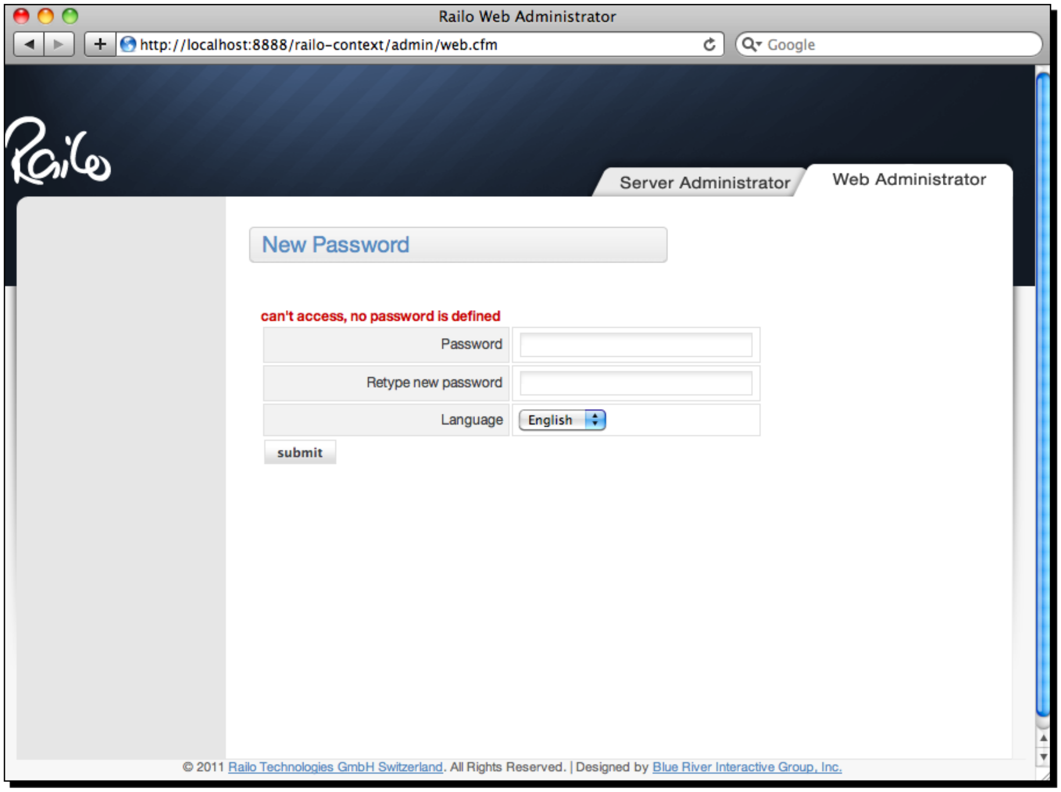
You just downloaded and ran Railo Express. Nothing was installed to your computer, but you're still able to get up and running with Railo extremely quickly.

Customizing Railo Express

Now that you have Railo running, let's customize it a bit. We will need to assign the administrators some passwords, add our own CFML files, and start developing!

Time for action – setting the administrator's password

1. To update the passwords for both the server and the web administrators, let's point our browser to `http://localhost:8888/railo-context/admin/web.cfm`. You will get the web administrator's login screen. Because no password is currently set, Railo Server will prompt you to set one:



can't access, no password is defined

Password	
Retype new password	
Language	English

submit

© 2011 Railo Technologies GmbH Switzerland. All Rights Reserved. | Designed by Blue River Interactive Group, Inc.

2. Enter your password and retype it to set the web administrator password.
3. Now click on the **Server Administrator** tab at the top-right-hand side and repeat the same procedure again. You have now set the passwords for the whole server administrator and for the web administrator.
4. Now that we have secured our server, we can start coding. You can find the webroot in `<railo folder>/webroot`, which is where we will save our template files. The main file will be `index.cfm`, which is already there.

What just happened?

You now have a fully-functional development environment for Railo and you didn't even have to install any software to get there!

Running the Railo Server Tomcat installer

The Railo Server Tomcat installer was developed with several goals in mind. Those goals are as follows:

- **Easy Integration with the existing web servers**: As a general rule, most developers are comfortable working with either Apache or IIS as their web server, and don't want or need to use anything different. The Railo Server Tomcat installer provides a way for those developers to get up and running quickly with support for their preferred web servers. This means those developers (and possibly you) will not have to change their development style too much in order to get comfortable with Railo.
- **Better "Control Panel" support**: By seamlessly supporting existing web servers, the Railo installer can very easily be deployed by hosting companies or individuals using services from a hosting company. This gives Railo developers the ability to use whatever Control Panel (such as cPanel, Plesk, Virtualmin, and so on) they're most comfortable with and still have Railo support.
- Easy Support for multiple sites.
- The Railo installer makes it easy to run many sites, and many web applications off a single instance of Railo. This is different from, say, a WAR deployment where possibly many instances of Railo can be deployed at a single time. It also saves memory because there is only one instance of Railo running.

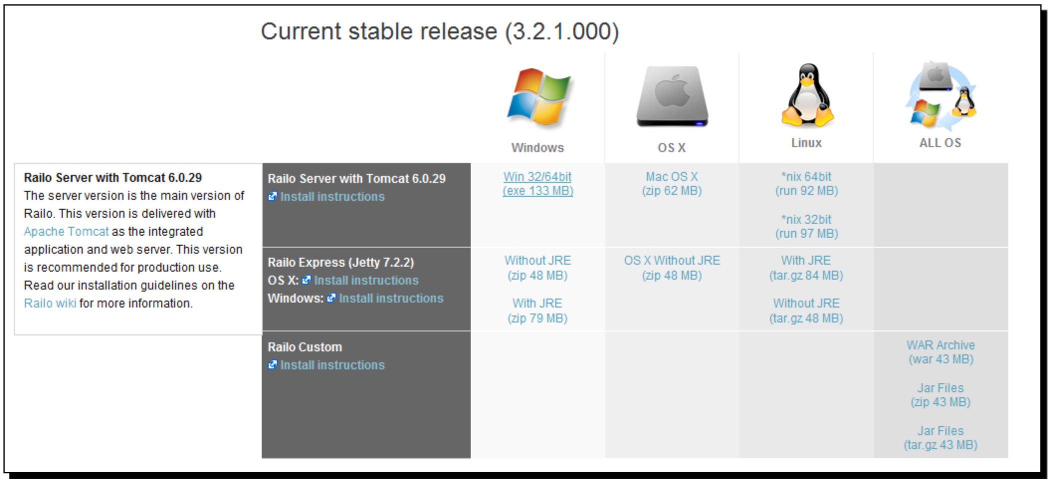
The Railo installers provide the most flexible and easy ways to get up and running on Railo.

Time for action – installing on Windows

Let's install Railo Server on a Windows machine to see how easy it is.

Before we install this version, make sure that Railo Express is not running anymore. If you still have the `Start/start.bat` console window open, close it now. The reason for this is that both Jetty (Railo Express) and Tomcat listen to port 8888 by default, and only one program can bind them to a given port at the same time.

1. As with Railo Express, let's head to `http://www.getrailo.org/index.cfm/download/`
2. This time, let's download the **Railo Server with Tomcat** version:

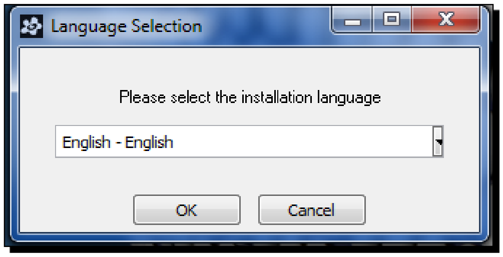


Current stable release (3.2.1.000)

Railo Server with Tomcat 6.0.29
The server version is the main version of Railo. This version is delivered with Apache Tomcat as the integrated application and web server. This version is recommended for production use. Read our installation guidelines on the Railo wiki for more information.

	Windows	OS X	Linux	ALL OS
Railo Server with Tomcat 6.0.29 Install instructions	Win 32/64bit (exe 133 MB)	Mac OS X (zip 62 MB)	*nix 64bit (run 92 MB) *nix 32bit (run 97 MB)	
Railo Express (Jetty 7.2.2) OS X: Install instructions Windows: Install instructions	Without JRE (zip 48 MB) With JRE (zip 79 MB)	OS X Without JRE (zip 48 MB)	With JRE (tar.gz 84 MB) Without JRE (tar.gz 48 MB)	
Railo Custom Install instructions				WAR Archive (war 43 MB) Jar Files (zip 43 MB) Jar Files (tar.gz 43 MB)

3. Once we have downloaded the executable, we run it, We select the language we want to install it in, and click on **OK**:

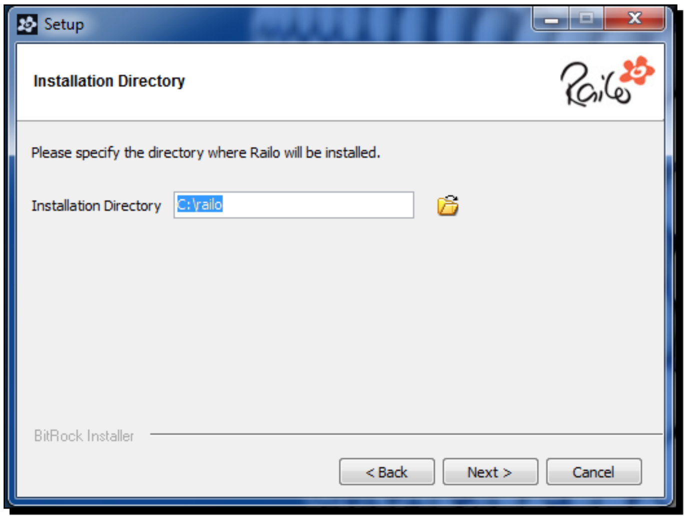


Please select the installation language

English - English

4. When we get to the introduction screen, we click on **Next >**.

5. When asked where to install Railo Server, we leave it as `C:\railo` and click on **Next >**:



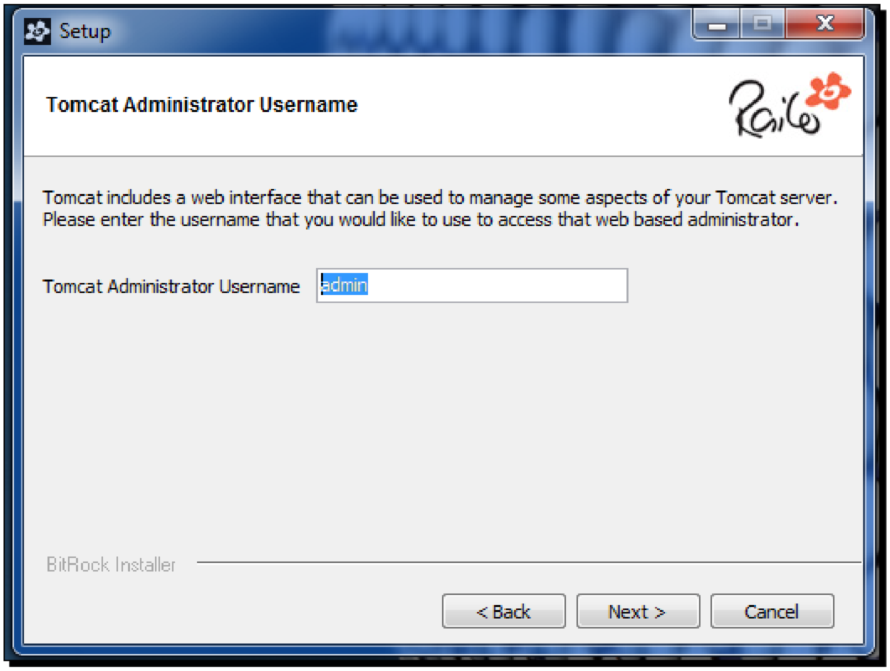
Setup

Installation Directory

Please specify the directory where Railo will be installed.

Installation Directory: C:\railo

6. The next screen asks us what we would like the Tomcat Administrator's username to be. We can leave it as `admin` and click on **Next >**:



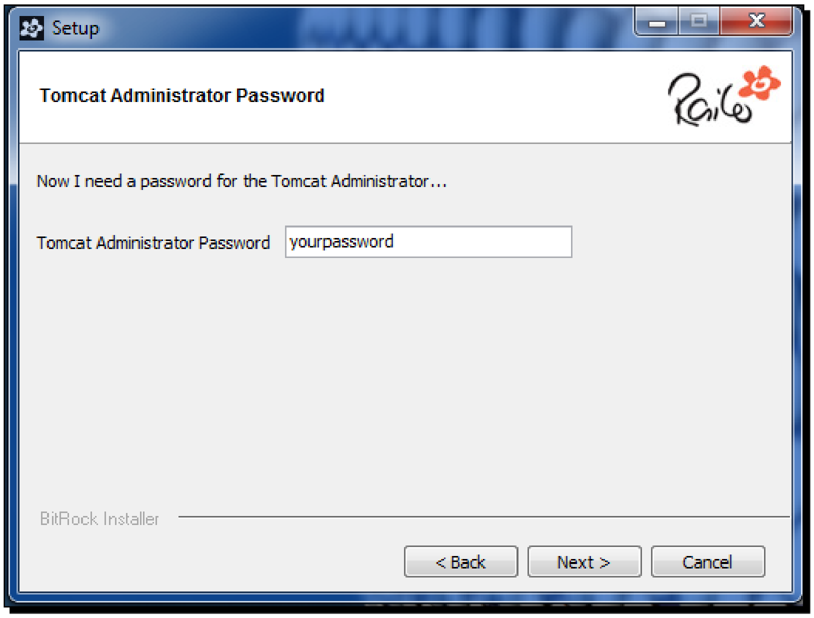
Setup

Tomcat Administrator Username

Tomcat includes a web interface that can be used to manage some aspects of your Tomcat server. Please enter the username that you would like to use to access that web based administrator.

Tomcat Administrator Username: admin

7. The **Tomcat Administrator Password** screen comes up. We then enter the password we want for it and click on **Next >**.

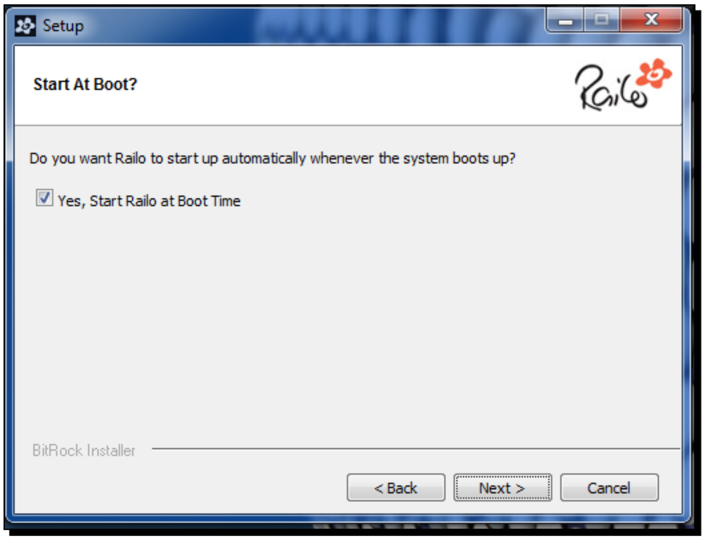


Tomcat Administrator Password

Now I need a password for the Tomcat Administrator...

Tomcat Administrator Password: yourpassword

8. The **Start at Boot??** screen appears and we leave it checked so that when the machine restarts, we know that Railo Server will be ready to serve requests. We click on **Next>**

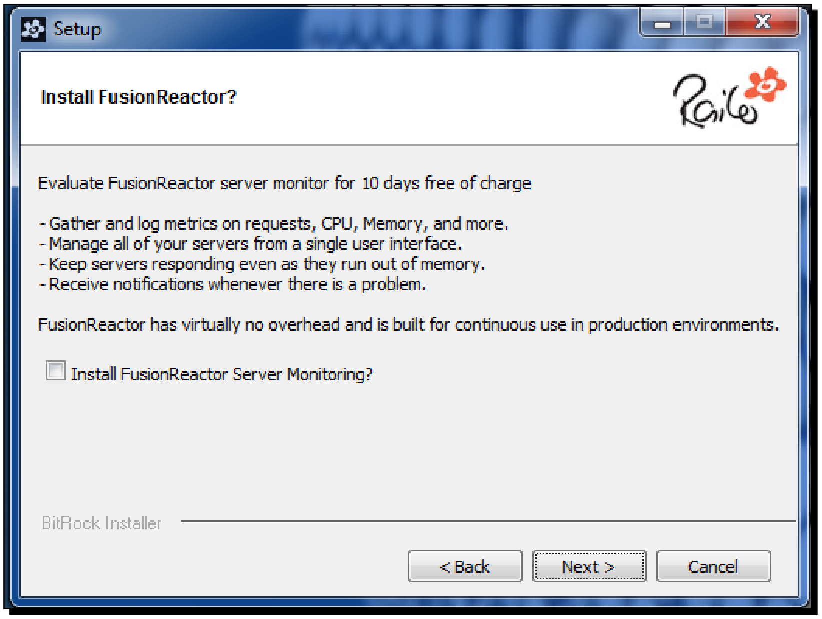


Start At Boot?

Do you want Railo to start up automatically whenever the system boots up?

☑ Yes, Start Railo at Boot Time

9. The **Install FusionReactor** screen appears. **FusionReactor** is a server-monitoring tool by Intergral GmbH. It is outside the scope of this book, so for this example, we will leave it unchecked and click on **Next>:**



Install FusionReactor?

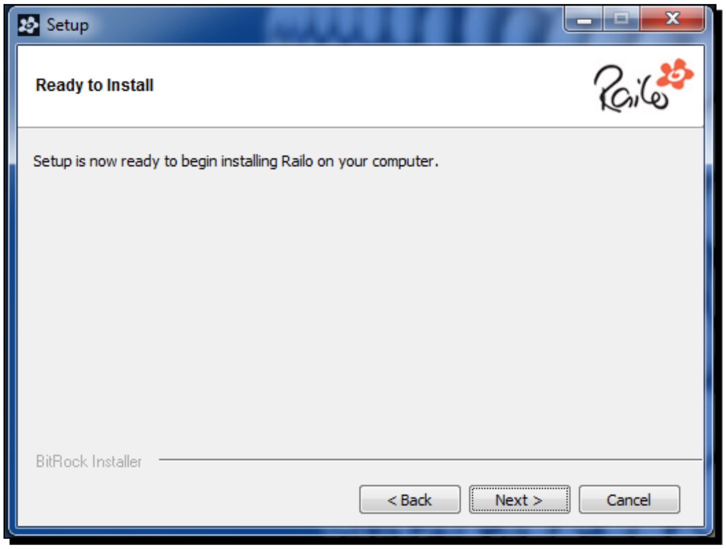
Evaluate FusionReactor server monitor for 10 days free of charge

- Gather and log metrics on requests, CPU, Memory, and more.
- Manage all of your servers from a single user interface.
- Keep servers responding even as they run out of memory.
- Receive notifications whenever there is a problem.

FusionReactor has virtually no overhead and is built for continuous use in production environments.

□ Install FusionReactor Server Monitoring?

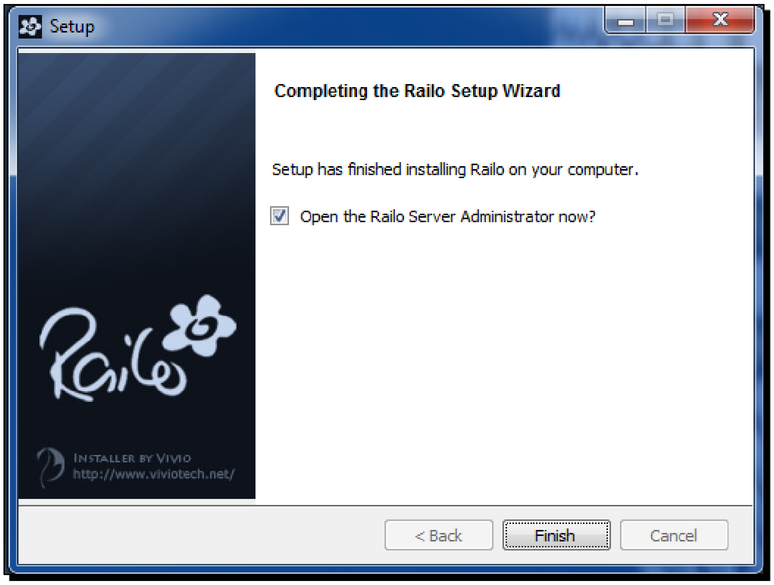
10. Now that the Railo Server installer has gathered all the information it needs, it is ready to install Railo. Click on **Next >**:



Ready to Install

Setup is now ready to begin installing Railo on your computer.

11. After a few seconds, you should get a confirmation screen that everything has been installed and you have the option to open the Railo Server Administrator. Leave this checked and click on **Finish**:



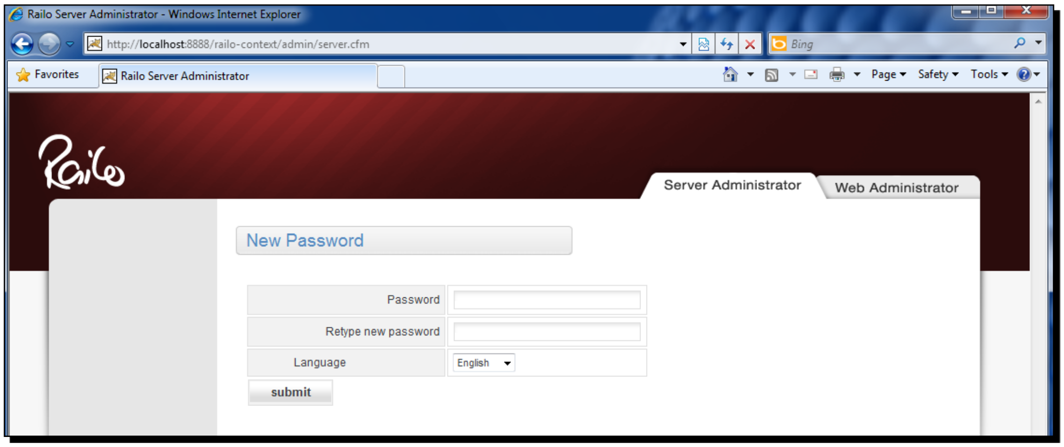
Setup

Completing the Railo Setup Wizard

Setup has finished installing Railo on your computer.

☑ Open the Railo Server Administrator now?

12. You should now be taken to the welcome screen of Railo Server. Just as with Railo Express, you can go to `http://localhost:8888/railo-context/admin/web.cfm` to change the server administrator and web administrator's password:



Railo Server Administrator - Windows Internet Explorer

http://localhost:8888/railo-context/admin/server.cfm

New Password

Password	
Retype new password	
Language	English

13. Congratulations! Railo is now installed on your computer!

What just happened?

You now have Railo Server installed on your computer, running under the Tomcat servlet engine. You can now create sites in IIS or Apache like you normally do, and have the CFML code that you put in each of those sites processed by Railo.

> **Differences in installers**
>
> If the installer had different steps for your installation it might be because you already have a web server, such as Apache or IIS, installed. You can follow the installation process for your particular settings and all that should change at the end of the process is the URL that you will be using.

Adding CFML-enabled sites to IIS7

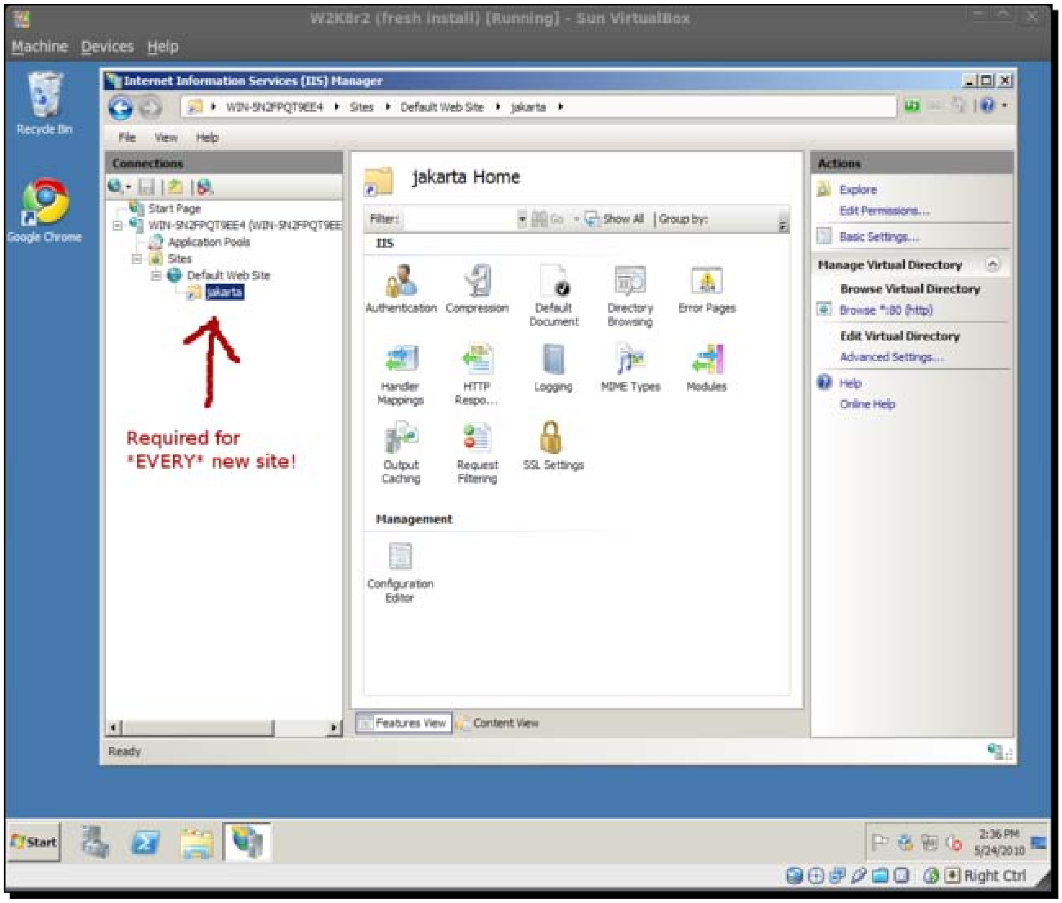
The Railo Server installer creates an instance of Railo Server that is built to support having a web server in front of it. In this section, we will review how to go about adding sites to IIS and configuring them to process CFML code. The process is similar for both IIS and Apache, so we'll just go through the IIS process here.

Time for action – adding a site to IIS7

To add a Railo-enabled site to IIS7, just carry out the following steps:

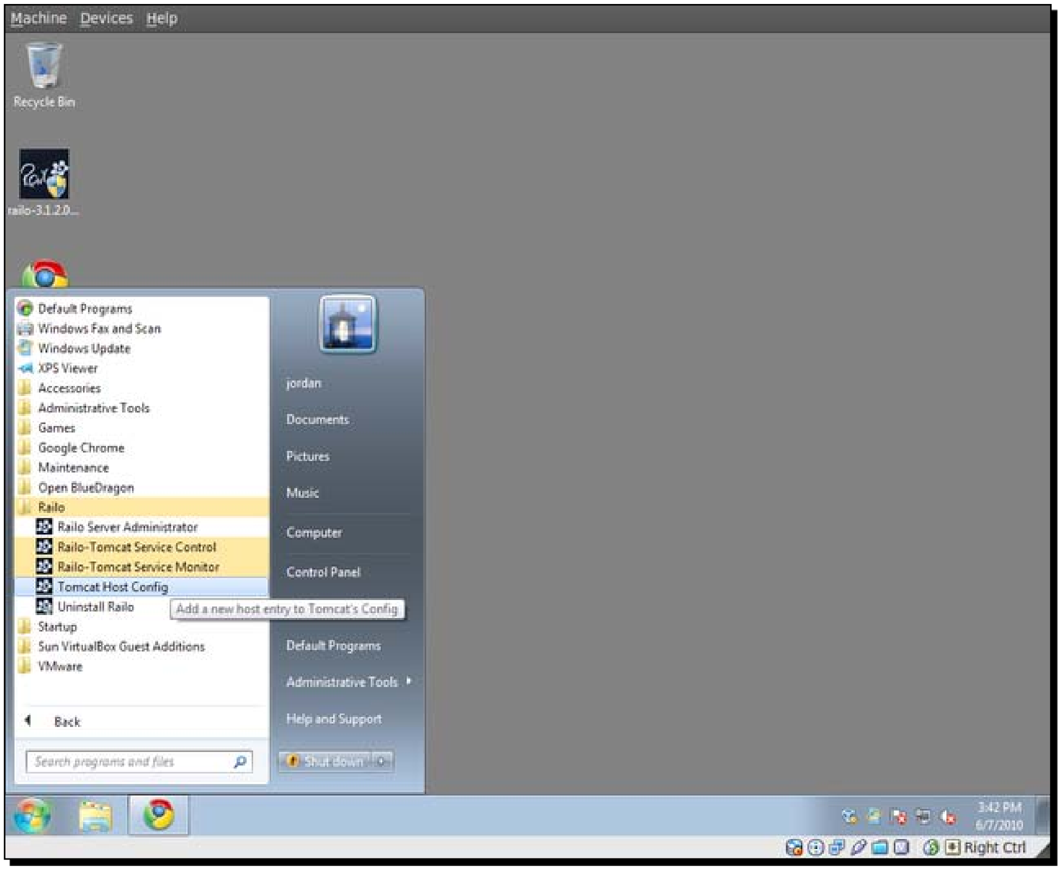
1. Create your site in IIS. This part is no different than what you would normally do to create a site in IIS.

2. In the IIS site that you want to connect, create a virtual directory in your new site named `jakarta` and point it to your connector directory, which is usually at `c:\railo\connector`.

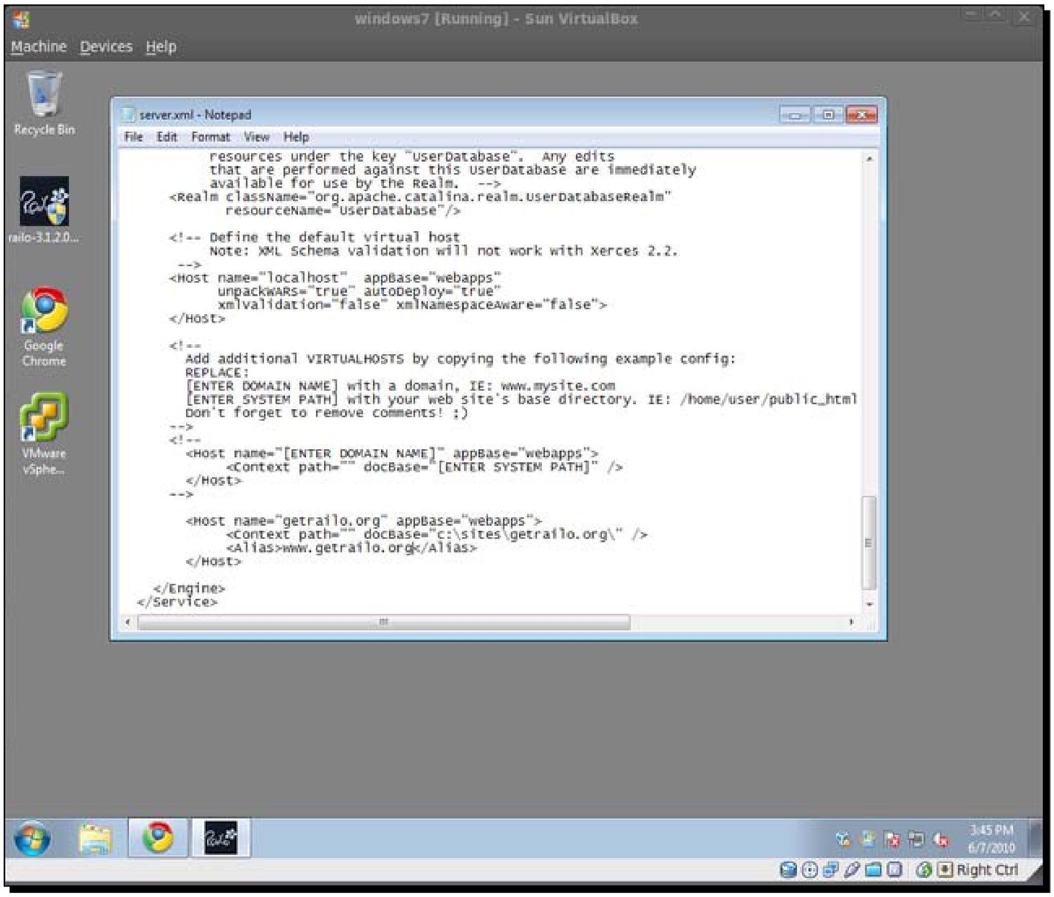


Required for *EVERY* new site!

3. Next, go to your **Start** menu. Click on the **Railo** folder, and then click on the **Tomcat Host Config** link. This will open the Tomcat's `server.xml` file in Notepad, so you can edit it.



4. Notice the comments in the `server.xml` file. Add an additional Host entry to the file that states your domain name and where your files are located for that domain.



```
        resources under the key "UserDatabase".  Any edits
        that are performed against this UserDatabase are immediately
        available for use by the Realm.  -->
   <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
          resourceName="UserDatabase"/>

   <!-- Define the default virtual host
        Note: XML Schema validation will not work with Xerces 2.2.
    -->
   <Host name="localhost"  appBase="webapps"
         unpackWARs="true" autoDeploy="true"
         xmlValidation="false" xmlNamespaceAware="false">
   </Host>

   <!--
     Add additional VIRTUALHOSTS by copying the following example config:
     REPLACE:
     [ENTER DOMAIN NAME] with a domain, IE: www.mysite.com
     [ENTER SYSTEM PATH] with your web site's base directory. IE: /home/user/public_html
     Don't forget to remove comments! :)
   -->
   <!--
     <Host name="[ENTER DOMAIN NAME]" appBase="webapps">
          <Context path="" docBase="[ENTER SYSTEM PATH]" />
     </Host>
   -->

     <Host name="getrailo.org" appBase="webapps">
          <Context path="" docBase="c:\sites\getrailo.org\" />
          <Alias>www.getrailo.org</Alias>
     </Host>

  </Engine>
</Service>
```

5. Now that you've set up IIS and configured your new host in Tomcat, you need to restart Tomcat for your changes to take effect. You can do this using the **Tomcat Service Control** option in the **Railo** Start menu folder.

Getting up and running with the Railo WAR and Jetty

The Railo WAR is a general-purpose install method that is meant for use with any Java servlet container (such as Jetty, Tomcat, JBoss, Resin, and so on) that supports WAR deployments. If your organization is already using Java and a Java Servlet Engine, this will be an easy way to get an application up and running within your existing Java environment.

What is a WAR?

In the context of Java Servlet Engines, a WAR file is a file that contains all the programs and classes that Java needs in order to run a single application. For Railo, this means that the WAR file contains all the programs and classes needed to run Railo and process CFML code.

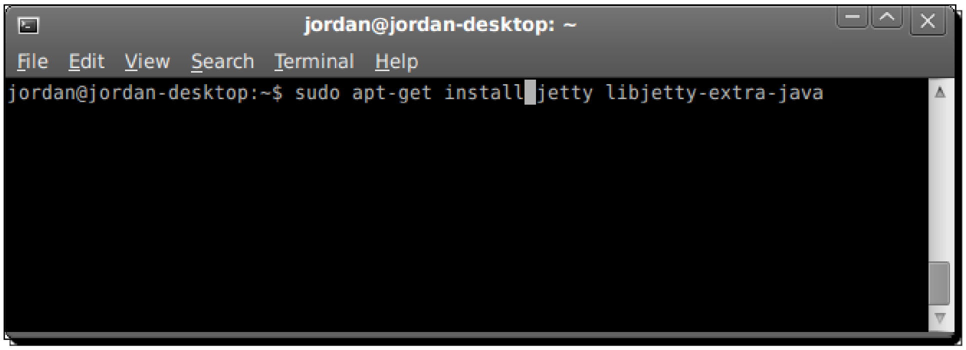
WAR files can be deployed anywhere, on any operating system. Earlier in the chapter we covered installing to the Tomcat Servlet Engine under Windows OS. We will now cover installing Railo to a Jetty Servlet Engine running on top of an Ubuntu Linux machine.

Time for action – downloading and installing Jetty

Let's download and install Jetty so we can get Railo deployed:

1. Open a terminal window in Ubuntu by going to **Applications** | **Accessories** | **Terminal**.
2. Once you have a **Terminal** open, type in the following:

```
$ sudo apt-get install jetty libjetty-extra-java
```



```
jordan@jordan-desktop:~$ sudo apt-get install jetty libjetty-extra-java
```

3. Ubuntu will prompt you for your password so that it knows you have the permission to be installing software on the machine. Go ahead and enter in your password and Ubuntu will install Jetty from there.

What just happened?

You just used the Ubuntu command `apt-get` to tell Ubuntu to go out to a public file location (called a `repository`), download the files for Jetty, and install them on your computer. It's actually a pretty complicated process, but it's made really simple with this one command.

Time for action – booting up Jetty

It is important to note that Jetty will not start yet. There is a safety precaution in a Jetty configuration file that helps avoid problems. We need to go edit the default Jetty configuration and remove this safety block.

You can use whatever text editor you prefer, but we're going use the "vi" editor for this example.

Let's open up the `/etc/default/jetty` configuration file in the "vi" editor. To do this, all you have to do is type:

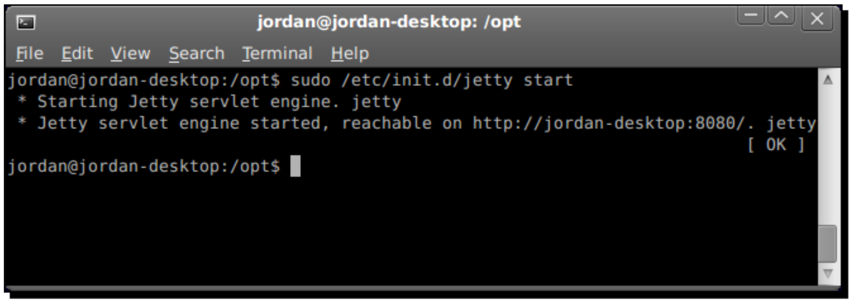
```
sudo vi /etc/default/jetty
```

1. Now that you're in "vi" editor, use the arrow keys and move your cursor to the beginning of the line that states **NO_START=1**.
2. Hit the *I* key on your keyboard. This will put the "vi" editor in what's called *Insert* **mode** and allow you to insert text. You will see a white **- Insert -** at the bottom-left-hand side of your screen. This is how you know you're in Insert mode.
3. Type a pound sign (*#*) in front of the "**NO_START=1** line, so that it looks like "**#NO_START=1**. This will comment out that line.
4. Hit the Escape key (*Esc*) to leave Insert Mode. You will see that the white **- Insert -** is no longer at the bottom-left of your screen.
5. Now type `:wq!` and hit the *Enter* key. This will save your file and exit the "vi" editor. You're done!

Now that we've edited the configuration file, we're free to start up Jetty. Type the following to start the Jetty Servlet Engine:

```
$ sudo /etc/init.d/jetty start
```

You will see Jetty start up, which is something like this:

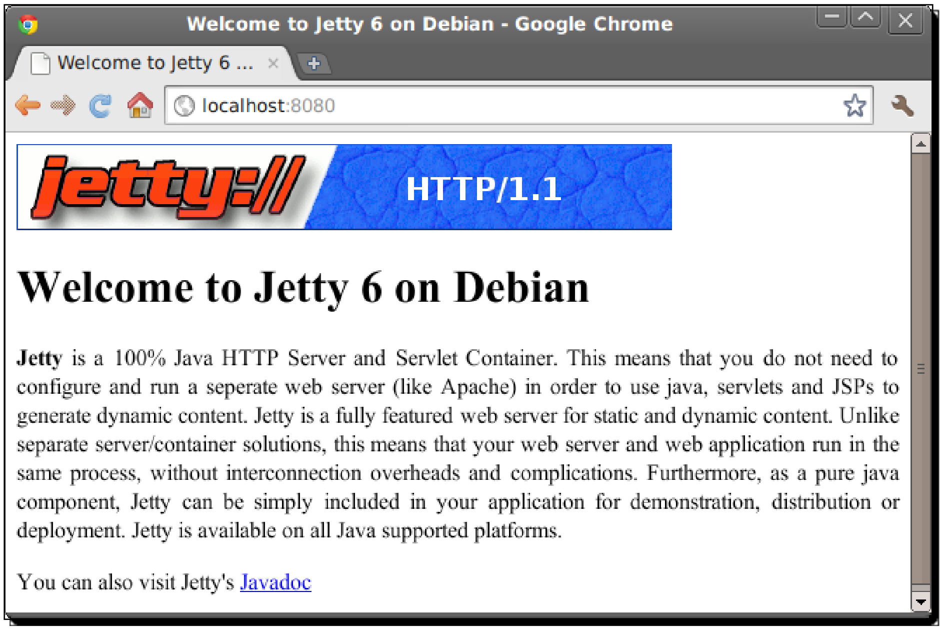


jordan@jordan-desktop: /opt
File Edit View Search Terminal Help

```
jordan@jordan-desktop:/opt$ sudo /etc/init.d/jetty start
 * Starting Jetty servlet engine. jetty
 * Jetty servlet engine started, reachable on http://jordan-desktop:8080/. jetty
                                                                        [ OK ]
jordan@jordan-desktop:/opt$ 
```

This screen is telling you that Jetty was able to start and is now listening on port **8080** of your local computer. My computer name is **jordan-desktop.** This is what the script put as the URL. Instead of `jordan-desktop`, it's probably better to use `localhost`, because it's less likely to be firewalled and cause problems. If you're installing to a remote machine, you could also use that machine's remote IP address. Just make sure there's no firewall blocking port **8080**. We will need that port open in order to use it.

Now, let's test to make sure we can see our new Jetty install. Try hitting `http://localhost:8080/` and make sure you get the Jetty page, as shown in the following screenshot:



Welcome to Jetty 6 on Debian

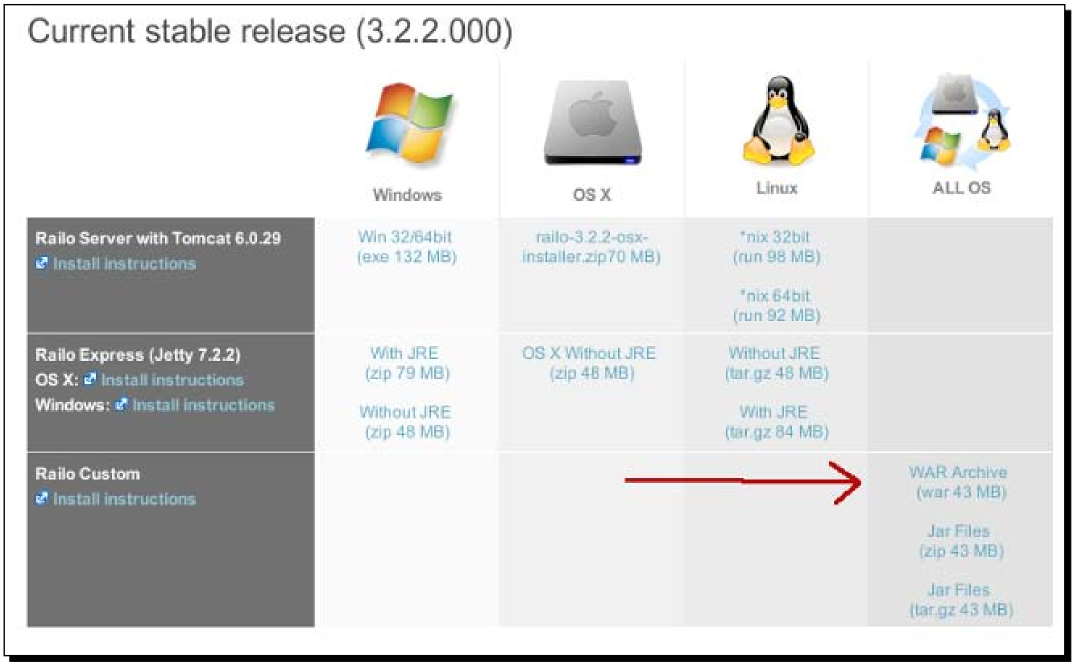
Jetty is a 100% Java HTTP Server and Servlet Container. This means that you do not need to configure and run a seperate web server (like Apache) in order to use java, servlets and JSPs to generate dynamic content. Jetty is a fully featured web server for static and dynamic content. Unlike separate server/container solutions, this means that your web server and web application run in the same process, without interconnection overheads and complications. Furthermore, as a pure java component, Jetty can be simply included in your application for demonstration, distribution or deployment. Jetty is available on all Java supported platforms.

You can also visit Jetty's Javadoc

If you see the same thing as in the previous screenshot, you're all set! Your computer is now prepared to install the Railo WAR.

Time for action – downloading and deploying the Railo WAR

The process of deploying a WAR file is super simple. You just download the WAR file, place it in the `webapps` or `webapp` folder of whichever servlet engine you're using, and your servlet engine will handle the rest via `AutoDeploy`. The following steps describe how to accomplish this with our Ubuntu/Jetty install.



Current stable release (3.2.2.000)

	Windows	OS X	Linux	ALL OS
Railo Server with Tomcat 6.0.29 Install instructions	Win 32/64bit (exe 132 MB)	railo-3.2.2-osx-installer.zip70 MB)	*nix 32bit (run 98 MB) *nix 64bit (run 92 MB)	
Railo Express (Jetty 7.2.2) OS X: Install instructions Windows: Install instructions	With JRE (zip 79 MB) Without JRE (zip 48 MB)	OS X Without JRE (zip 48 MB)	Without JRE (tar.gz 48 MB) With JRE (tar.gz 84 MB)	
Railo Custom Install instructions				WAR Archive (war 43 MB) Jar Files (zip 43 MB) Jar Files (tar.gz 43 MB)

1. Move to the Jetty's **webapps** folder. You can do that in Ubuntu's **Terminal** interface by typing in the following command:

```
$ cd /var/lib/jetty/webapps/
```

2. Download Railo's WAR file. At the time of this writing, the most recent Railo release is 3.2.2.000, so let's download that WAR to our Jetty `webapps` directory.

3. To save time in the future, let's rename that WAR file to simply `railo.zip` with the following command:

```
$ sudo mv railo-3.2.2.000.war railo.zip
```

4. Unzip it so that it's deployed:

```
$ sudo unzip -d railo railo.zip
```

5. Remove the ZIP file as we're done with it:

```
$ sudo rm railo.zip
```

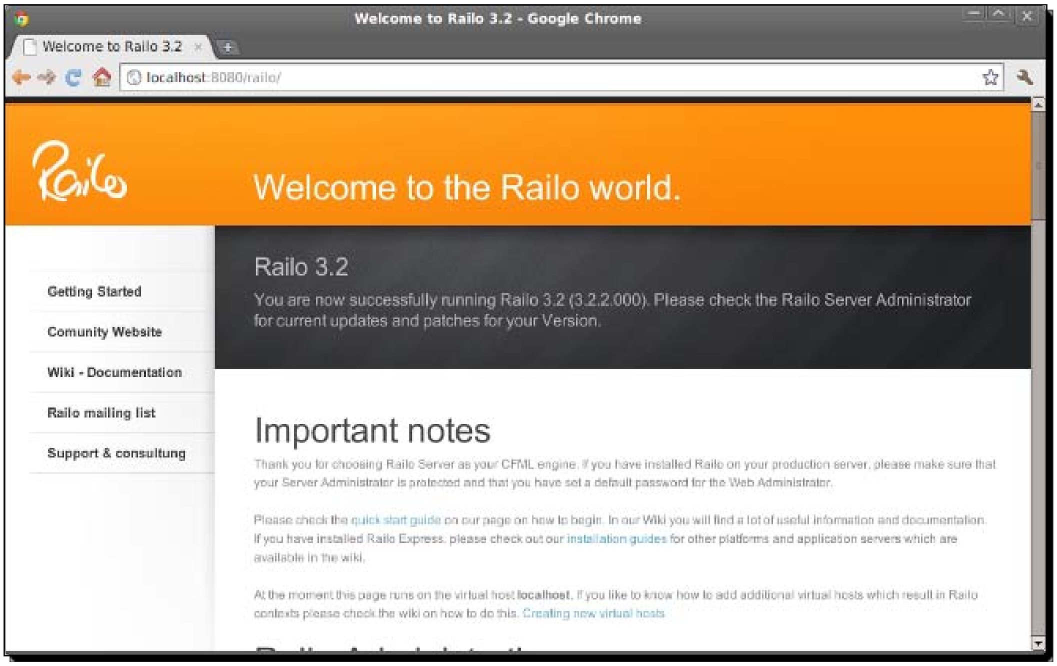
6. Finally, change the permissions of the extracted files to match that of Jetty:

```
$ sudo chown -R jetty:adm railo
```

7. Now let's restart Jetty so it will pick up our new `railo` folder:

```
$ sudo /etc/init.d/jetty restart
```

8. Now we should be able to access Railo from the web at `http://localhost:8080/railo/`.



Welcome to the Railo world.

Railo 3.2

You are now successfully running Railo 3.2 (3.2.2.000). Please check the Railo Server Administrator for current updates and patches for your Version.

Important notes

Thank you for choosing Railo Server as your CFML engine. If you have installed Railo on your production server, please make sure that your Server Administrator is protected and that you have set a default password for the Web Administrator.

Please check the quick start guide on our page on how to begin. In our Wiki you will find a lot of useful information and documentation. If you have installed Railo Express, please check out our installation guides for other platforms and application servers which are available in the wiki.

At the moment this page runs on the virtual host **localhost**. If you like to know how to add additional virtual hosts which result in Railo contexts please check the wiki on how to do this. Creating new virtual hosts

9. Congratulations! We have installed Railo with Jetty Web Server.

Once you see this screen, you're all set! You are now able to get started developing on Railo under Jetty.

To add your own CFML files to this install, you just add them to the `railo` directory. Be sure to not touch the `WEB-INF` directory, as that's where Railo lives and processes your CFML files.

What just happened?

You just installed Railo on top of your Jetty install. You can now drop your own CFML files into the railo directory. Be sure to leave the WEB-INF directory there. It contains important files that Railo uses to process your CFML code. But don't worry, this folder is not web accessible.

Summary

In this chapter, we saw how simple it was to install Railo Server in various environments. We started with the Railo Express version, which includes Jetty and is standalone. It can be started from a simple script without needing to install or connect it to a web server. We then used the Railo Server installer to install Railo with Apache Tomcat into a Windows environment and then connected the IIS server to our sites. Finally, we saw how easy it was to install on an Ubuntu server by simply getting Jetty through the apt command and just deploying it to the Railo WAR.

We also briefly touched on securing the Railo Web and Server Administrators by adding a password and found out where we should put our CFML templates to get started in developing.

In the next chapter, we are going to talk more about CFML, so that we can get to grips with the structure and syntax of this very easy, yet powerful, language.

3

CFML Language

Now that we have got Railo Server installed, we can finally get down to some coding!

This chapter will introduce you to the CFML programming language, the programming language that lets you build awesome web applications that can run on Railo Server. The great thing about the CFML language is that it is really easy to get started, using simple tags and functions that generally are self descriptive, which will get you coding applications from the start and become a master of the language in no time at all!

In this chapter, we shall learn the following:

- Basics of the language: tags, functions, variables, operators, and scopes
- Database access: configuring data sources, running queries and stored procedures
- Handling web data such as forms, URL variables, cookies, and sessions
- Object-oriented programming with components

Why don't we dive in and get started!

> **The CFML language history**
>
> The CFML language was initially created by Allaire Technologies for their ColdFusion Server in 1995. CFML is an acronym for **ColdFusion MarkUp Language**, and currently three servers (Railo, OpenBD (Open Blue Dragon), and Adobe's ColdFusion Server) support the language.

Basics of the CMFL language

The idea behind the CFML language was to let web developers have a way to create server-side applications quickly and easily. To do this, it was designed to use coding metaphors that are familiar, such as using tags (like HTML tags) and functions (just like JavaScript functions).

If you have done HTML development, you will be used to using tags to describe the layout of your document. With CFML, you describe what you want to do on the server side, for example, saving the contents of a feedback form, display details of a product stored in a database, allow a user to login to your site, and so on.

When you write CFML code, it is not displayed in the browser, but rather parsed by the server, providing you a way to output some code. To show you an example, let's try the quintessential **Hello World** example.

Time for action – Hello World!

Create a file called `listing3_1.cfm` with the following contents (most CFML pages end with the extension `.cfm`.. This tells Railo Server that this file should be parsed):

```
<html>
  <head><title>Example 1</title></head>
  <body>
    <cfset hello = "Hello World">
    <cfoutput>#hello#</cfoutput>
  </body>
</html>
```

You can now save this file to `<railo install directory>/webroot/listing3_1.cfm`, and then we can view the results by going to `http://localhost:8888/listing3_1.cfm` (you should be using the Railo Express edition you installed in *Chapter 2*, *Installing Railo Server* for all these examples). You will see the output "Hello World" displayed in your browser.

What just happened?

In the preceding code, we have an example HTML document that we have added some CFML code to. In the preceding example, we used the `<cfset>` tag to set the variable `hello` with a value of `"Hello World"`. Then we use the `<cfoutput>` tag to say that we want to start outputting the `hello` variable to the document.

We use the # (pound or hash) sign surrounding a variable to output it. This is one of the most basic ways of outputting variables.

If you look at the source of the browser document, you can see that the server has replaced all the CFML tags and variables and output the following HTML:

```
<html>
  <head><title>Example 1</title></head>
  <body>

    Hello World
  </body>
</html>
```

CFML tags

There are a number of CFML tags that allow you to code nearly anything you can imagine. But before we start investigating some of these tags and their functions, it is good to understand the general syntax of CFML tags. The overall syntax of CFML tags can be described as:

```
<tagname attribute="value">
  Code/text that is affected by the surrounding tags.
</tagname>
```

A tag can have one or more attributes that have values similar to HTML. They can also wrap some code or text that will be affected by the surrounding tags. It is important to note that despite the previous example, all CFML tag names start with "cf"; so, for example, we have tags called `<cfabort>` , `<cfloop>`, `<cfoutput>`, and so on.

Single tags with attributes

A number of CFML tags can be used as a single tag, without the need to close it.
For example:

```
<cfabort>
```

You might notice that since it is a single tag, you don't need to add a closing tag. For the sharp eyed ones amongst you, you will notice that you also don't need to use XML-type syntax and add a closing forward slash in the tag like `<cfabort />` because CFML doesn't require them, but you can use them in your code. As an aside, the above tag stops the page execution. We shall look in detail at the functionality of each tag later on.

Tags can also have attributes, which define what they do as shown next:

```
<cffile action="read" file="/somefile.txt" variable="myFile">
```

The previous tag has a number of attributes that define how it functions better, in this case, to read a file from /somefile.txt and put it into a variable called myFile. Nearly all tags in the CFML language take some attributes.

Tags with expressions

As we saw before, there are tags that take attributes, but some tags just take an expression; two notable tags in this category are <cfset> and <cfif>. You already saw how you can set a variable with <cfset>, which is its purpose, but you can also evaluate expressions, for example:

```
<cfset ArrayAppend(myArray, "something")>
```

We use a function to add an item to the array myArray with the value something. There is no value set, but we have still achieved a goal.

Another tag that uses an expression instead of an attribute is the <cfif> tag. This tag allows you to perform conditional logic, depending on whether an expression is true, for example:

```
<cfif isNumeric(myValue)>
  The value is numeric
</cfif>
```

The previous code evaluates the value myValue with the function isNumeric() to check whether it is a number value. If the expression evaluates to true, then the contents between <cfif> and </cfif> are displayed.

Time for action – single tag example

Let's create an example where we can use a few single tags.

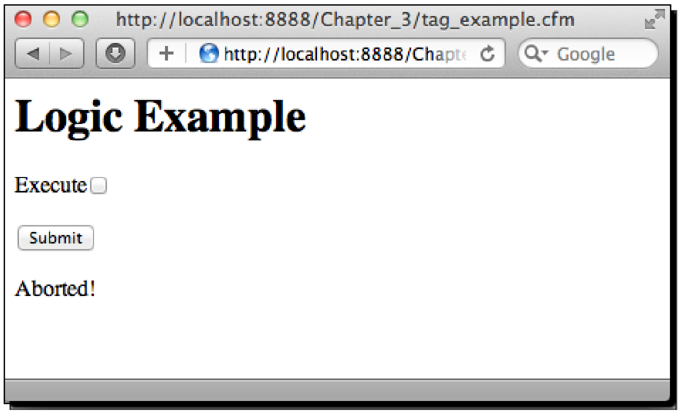
In our <Railo Install Directory>/webroot/Chapter_3/ folder, let's create a file called tag_example.cfm.

In this tag, let's add a form and some code that will respond to changes in a checkbox:

```
<cfparam name="FORM.doexecute" default="false">

<h1 id="logic_example">Logic Example</h1>
<form action="tag_example.cfm" method="post">
  <label for="execute">Execute</label><input
  type="checkbox" name="doexecute" value="true" id="execute">
  <p><input type="submit" value="Submit"></p>
</form>
```

```
<cfset output = "">
<cfif FORM.doexecute>
    <cfset output = "We are going to execute some code!">
<cfelse>
  Aborted!
  <cfabort>
</cfif>

<cfoutput>#output#</cfoutput>
<br>
This is the end of the file
```

Let's run this in the browser by going to `http://localhost:8888/Chapter_3/tag_example.cfm. You will see the form with a single tickbox:`


http://localhost:8888/Chapter_3/tag_example.cfm
http://localhost:8888/Chapt
Google

Logic Example

Execute □

Submit

Aborted!

If we submit the form, without ticking the checkbox, you will always see the **Aborted!** message at the bottom.

If we now tick the **Execute** checkbox, you should see a message saying:

We are going to execute some code!

This is the end of the file

What just happened?

In our example we have used a few tags to set variables in our code. At the start of the file, we put a `<cfparam name="FORM.doexecute" default="false">` statement, which sets the `FORM.doexecute` variable to false, but only if it doesn't exist. This is important to note. Otherwise we would have to check if that form variable has been passed in. This is a neater way of coding, as it removes the need to check for variables all the time.

We then create a simple form with a checkbox. This form will actually post back to the `tag_example.cfm` file.

We then use the `<cfset output="">` to set up a variable called `output`, and then use the `<cfif>`, `<cfelse>` and `</cfif>` tags to test whether the value of the `FORM.doexecute` value (which is passed into the form) is set to true or to false.

If the value of `FORM.doexecute` is false, we then use the `<cfabort>` tag to escape the rest of the execution of the file.

Tags with content

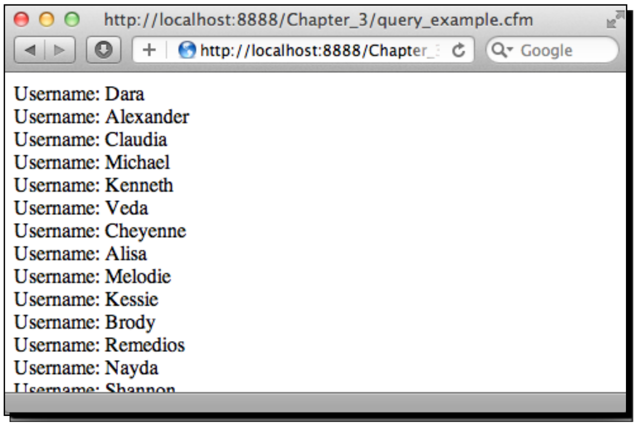
As you saw in the previous example, some tags take content, such as the `<cfif>` tag. There are other tags that can take parsed content within them. A notable example of this is the `<cfquery>` tag:

```
<cfquery name="myQuery" datasource="myDatabase">
  SELECT *
  FROM Users
</cfquery>
```

The code above shows how, instead of outputting the text within the `<cfquery>` tags, the content will be parsed as SQL and executed against the database defined in the `datasource` attribute. In this example, we would now have the results of our query in the `myQuery` variable. If we want to loop through the results, we can do the following:

```
<cfoutput query="myQuery">
   Username: #name#  <br>
</cfoutput>
```

The previous code would output a list of all the values in the `name` column of the `Users` table (of course, if that is what you have in your database).


http://localhost:8888/Chapter_3/query_example.cfm
http://localhost:8888/Chapter_
Google

```
Username: Dara
Username: Alexander
Username: Claudia
Username: Michael
Username: Kenneth
Username: Veda
Username: Cheyenne
Username: Alisa
Username: Melodie
Username: Kessie
Username: Brody
Username: Remedios
Username: Nayda
Username: Shannon
```

Tags with sub tags

There are a number of tags that can only be used within another tag. These tags usually allow the parent tag to filter or pass more variables. A great example is the `<cfqueryparam>` tag. This allows you to specify the value and type that you are passing to a query and allows you to stop SQL Injection attacks on your system. If we were looking for a specific user from our database table, we might do something like:

```
  <cfquery name="myQuery" datasource="myDatabase">
  SELECT *
  FROM Users
  WHERE userid = #URL.userid#
</cfquery>
```

The problem with the code is that the variable (that we are passing through the URL) of `userid` could be hacked by a malicious user of the site. To make sure that the variable passed into user ID is of the right format, and there are no SQL attacks, we can insert a `<cfqueryparam>`. For example:

```
    <cfquery name="myQuery" datasource="myDatabase">
      SELECT *
      FROM Users
      WHERE userid = <cfqueryparam cfsqltype="cf_sql_numeric"
value="#URL.userid#">
    </cfquery>
```

Now that we have added the `<cfqueryparam>` tag, you can see that we define the `cfsqltype` of the variable that we are going to pass, in this case, a `cf_sql_numeric` variable and the value from the URL. Railo uses a mapping of database column types to the `cf_sql_*` attribute variables so that the previous code can run on any database, rather than being database-type-specific. Other tags that take sub tags such as `<cfqueryparam>` for filtering purposes are `<cfhttp>` with `<cfhttpparam>`, `<cfmail>` with `<cfmailparam>`, and `<cfstoredproc>` with `<cfprocparam>`, to name a few. There is another type of tag that has to be placed within another tag, and those are some of the logic operation tags, such as `<cfcase>`, `<cfelse>`, and `<cfelseif>`. The `<cfcase>` tag can only go inside a `<cfswitch>` tag, for example:

```
<cfswitch expression="#URL.action#">

  <cfcase value="showpage">
    <!---Show the page-à
  </cfcase>

  <cfcase value="editpage">
    <!---Edit the page -à
  </cfcase>
```

```
    <cfdefaultcase>
      <!---Do some other action -à
    </cfdefaultcase>
  </cfswitch>
```

We now have a parent `<cfswitch>` tag in which we check the value of the `URL.action` parameter passed to the page. Then we have a series of cases that we can check with the `<cfcase>` tag, so if the value of `URL.action` is "`showpage`", in the case above, we would run the actions inside the `<cfcase value="showpage">` statement.

Another example is the `<cfelse>` and `<cfelseif>` tags, which could be used to write the previous function:

```
<cfif URL.action EQ "showpage">
  <!---Show the page -à

<cfelseif URL.action EQ "editpage">
  <!---Edit the page -à

<cfelse>
  <!--- Do some other action -à
</cfif>
```

As you have seen, CFML tags are easy to get started with. If you have ever developed HTML pages, you should get a grasp of using them in no time at all.

CFML functions

Functions in CFML come in two flavors, namely, the built-in functions that Railo provides as a part of the core engine, which are highly optimized, and the functions that you can write in CFML and have your code use. You have already seen the use of a built-in function with the use of `ArrayAppend()` and `isNumeric()`. So let's go and look at some functions in greater detail.

Time for action – using built-in functions

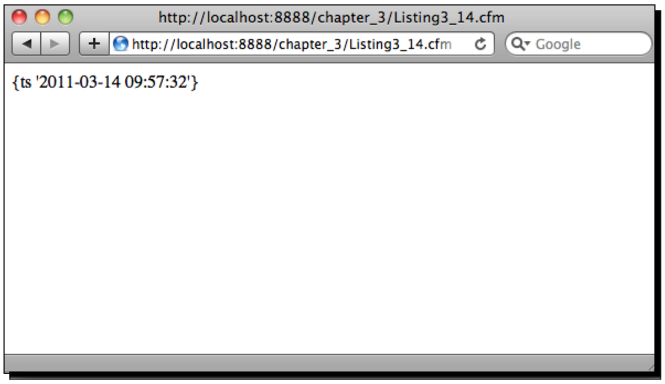
Railo has a large number of built-in functions (or BiFs) that provide a vast array of functionality at the tip of your fingers. On last count, there were roughly 500 functions that are available to you and of course that would be too much to cover each one of them in this book.

Functions provide a way to check and manipulate simple data, as well as manipulating complex variable types-like structures, arrays and dates.

The simplest example of using a function would be to get the current date, so for example, let's create a file in `<Railo Install Directory>/webroot/Chapter_3/` called `Listing3_14.cfm` and put the following code:

```
<cfoutput>
  #Now()#
</cfoutput>
```

By going to `http://localhost:8888/Chapter_3/Listing3_14.cfm`, your page will be displayed as:

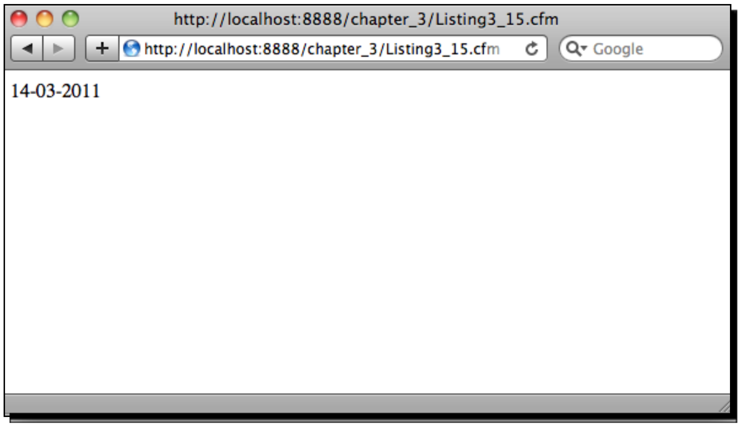

http://localhost:8888/chapter_3/Listing3_14.cfm
http://localhost:8888/chapter_3/Listing3_14.cfm
Google

```
{ts '2011-03-14 09:57:32'}
```

Many functions take a number of parameters. Since we are talking about dates, let's format the date a bit better with another function:

```
<cfoutput>
  #DateFormat(Now(), "dd-mm-yyyy")#
</cfoutput>
```

Now our date has been formatted nicely.


http://localhost:8888/chapter_3/Listing3_15.cfm
http://localhost:8888/chapter_3/Listing3_15.cfm
Google

```
14-03-2011
```

What just happened?

With Railo Server, you can also name the parameters as a name and value pairs with the "=" operator, rather than just passing them as a list, so the above function call can also be written as:

```
<cfoutput>
  #DateFormat(date=Now(), mask="dd-mm-yyyy")#
</cfoutput>
```

You can also use ":" as the operator between function parameters, so the code above can also be written as:

```
<cfoutput>
  #DateFormat(date:Now(), mask:"dd-mm-yyyy")#
</cfoutput>
```

The choice is up to you of course, since functionally, there is no difference, but it can improve clarity when reading a function. As you have seen in the previous code examples, functions can be used within functions, they can also be used inside the attributes of tags, for example:

```
<cfcookie name="userid" value="1" expires="#CreateDate(2011,12,29)#">
```

User-defined functions

Even though there are swathes of functions available to you with Railo Server; there are many times when you need to create your own functions to do something specific.

These could be functions that are not as generic as the ones provided by Railo Server and are specific to your use case, such as running a Regular Expression over a string or (as we shall see later) capitalizing the first letter of a word.

Functions that are not part of the core functionality of Railo Server are called User-defined functions and are pretty easy to create.

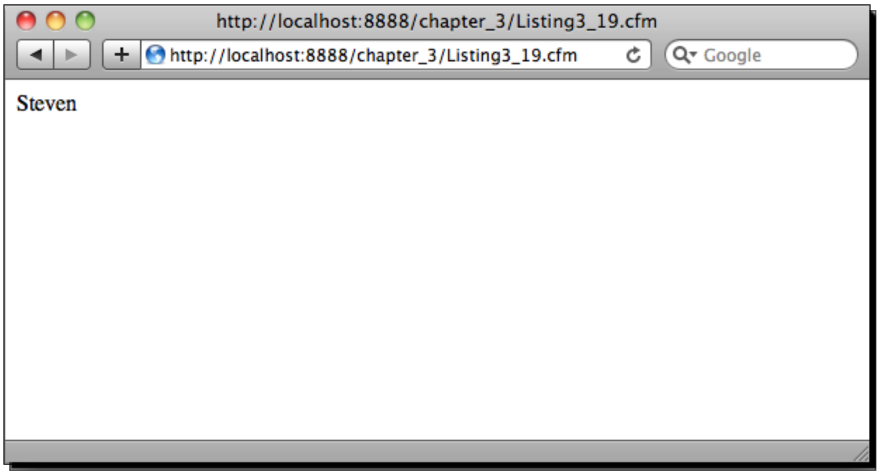
Time for action – using user-defined functions

To create a user-defined function all you have to do is use the `<cffunction>` tag. For example:

```
    <cffunction name="MakeCapitalised">
      <cfargument name="wordToCapitalise" type="string">
        <cfset FirstLetter = Left(wordToCapitalise,1)>
        <cfset FirstLetter = UCase(FirstLetter)>
        <cfset RestOfWord = Mid(wordToCapitalise, 2,
Len(wordToCapitalise)-1)>
```

```
        <cfreturn FirstLetter & RestOfWord>

      </cffunction>

        <cfoutput>
          #MakeCapitalised("steven")#
        </cfoutput>
```

The previous code will return the following:


http://localhost:8888/chapter_3/Listing3_19.cfm
http://localhost:8888/chapter_3/Listing3_19.cfm
Google

Steven

What just happened?

Let's work through the code, we first define the function with a name of `MakeCapitalised`:

```
        <cffunction name="MakeCapitalised">
```

Then we define the argument that it takes; in this case, we call it `wordToCapitalise` and say that it is of type string:

```
        <cfargument name="wordToCapitalise" type="string">
```

Next, we get the first letter of the `wordToCapitalise` using the `Left()` function, saying we want one character from the left of the `wordToCapitalise`:

```
        <cfset FirstLetter = Left(wordToCapitalise,1)>
```

Now we can convert that letter to uppercase with the `UCase()` function:

```
        <cfset FirstLetter = UCase(FirstLetter)>
```

Next we want to get the rest of the wordToCapitalise so that we can add it to the FirstLetter variable (which is now "S") using the Mid() function, which takes a string, the start, and the length of the string, to get:

```
        <cfset RestOfWord = Mid(wordToCapitalise, 2,
    Len(wordToCapitalise))>
```

And finally, we combine the FirstLetter and RestOfWord variables and return the output of the function with the <cfreturn> tag:

```
        <cfreturn FirstLetter & RestOfWord>
```

The final part of the code, we just call the function with the name **Steven** and the function is run. A word of warning though, you can name your functions anything you like, as long as you don't name it the same as the existing function in CFML. If you do this, you will get an error saying that it is already used by a built-in function.

CFML variables

We have already seen CFML variables at work in previous code examples, but what we have seen so far have been simple strings. The power of the CFML language is that you don't have to define what type variables are. Railo Server takes care of this and you can change the type of variable as you go along. So, for example:

```
<cfset a = "Mike">
<cfset a = 1>

<cfoutput>
   #a#
</cfoutput>
```

Will give result in "1". Variables in CFML are dynamically typed, so if we did the following:

```
<cfset a = 1>
<cfoutput>
       #isNumeric(a)#
</cfoutput>
```

We would get "true" being returned and displayed, the same would happen if we made "1" a string:

```
<cfset a = "1">
<cfoutput>
       #isNumeric(a)#
</cfoutput>
```

Variables can be simple types, such as strings, numbers and Booleans, but they can also be complex types. There are a number of complex variable types in CFML, such as Structures, Arrays and Queries. Functions and Components (see "Object Oriented Programming with Components" later in this chapter) can also be passed to variables.

Let's look at some of the more complex variables.

Structure variables

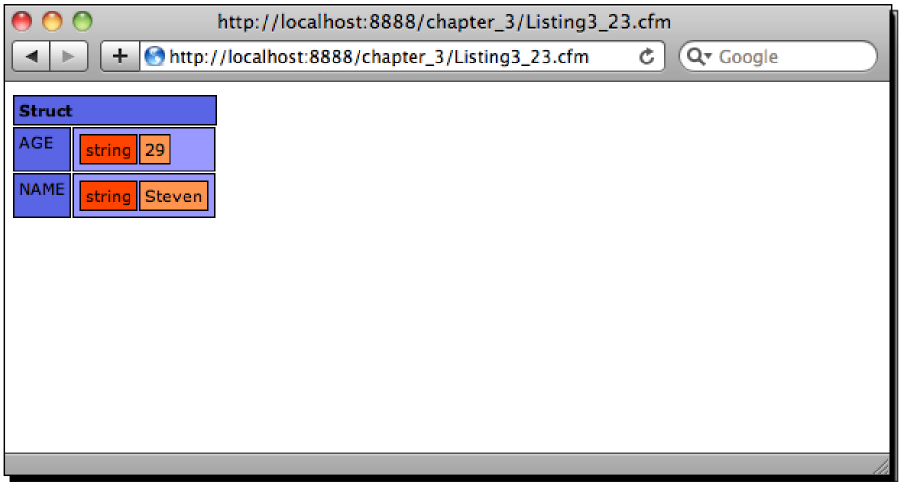
Structure variables are key/value representations of data. They are also known as Structs (or referred to as Maps or Collections). A structure can have as many keys as you want but they all must be unique, and can be referenced via the key name.

Time for action – using structures

Let's create an example so we can see it at work, create a file in your `<Railo Install Directory>/webroot/Chapter_3/structure_example.cfm` and put the following code in the template:

```
<cfset myStruct = StructNew()>
<cfset myStruct.name = "Steven">
<cfset myStruct.age = "29">

<cfdump var="#myStruct#">
```

So far we have been using `<cfoutput>` to display simple variables, but complex variables cannot just be displayed to the browser, hence we use the `<cfdump>` tag. This is a very handy tag to use to display complex values during development. It will produce the following output when you run your code:


http://localhost:8888/chapter_3/Listing3_23.cfm
http://localhost:8888/chapter_3/Listing3_23.cfm
Google

Struct		
AGE	string	29
NAME	string	Steven

If you want to reference the `name` key in the structure `myStruct` directly, you can do it in a couple of ways:

```
<cfoutput>
  #myStruct.name#
```

Or:

```
  #myStruct['name']#
</cfoutput>
```

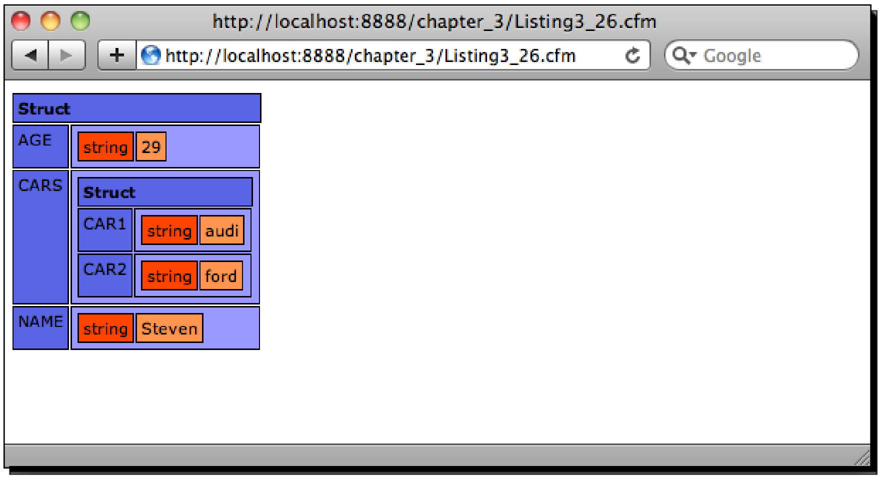
Also, you can create structures using implicit creation, using the curly brackets (**{}**) and the key valued objects directly:

```
<cfset myStruct = {name="Steven", age="29"}>
<cfdump var="#myStruct#">
```

Structures can contain any type of variable, but the keys need to be simple strings, so for example, we can create a structure within a structure:

```
<cfset myStruct = {name="Steven", age="29"}>
<cfset myStruct.cars = {car1="audi", car2="ford"}>
<cfdump var="#myStruct#">
```

It will give us the following:


http://localhost:8888/chapter_3/Listing3_26.cfm
http://localhost:8888/chapter_3/Listing3_26.cfm
Google

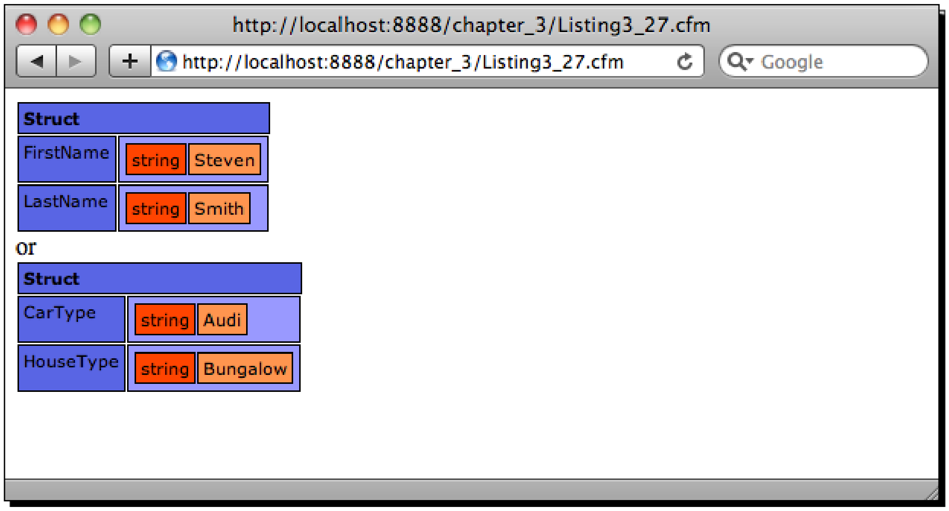
```
Struct
AGE    string 29
CARS   Struct
       CAR1   string audi
       CAR2   string ford
NAME   string Steven
```

You may have noticed that the keys of the structure are automatically made uppercase, this doesn't matter when you are accessing them, as CFML is case insensitive. If you need to make sure your keys maintain their case, you can do so by quoting the key name.
For example:

```
<cfset myStruct = {"FirstName"="Steven", "LastName"="Smith"}>
<cfdump var="#myStruct#">
```

Or:

```
<cfset myStruct2 = StructNew()>
<cfset myStruct2["CarType"] = "Audi">
<cfset myStruct2["HouseType"] = "Bungalow">
<cfdump var="#myStruct2#">
```


http://localhost:8888/chapter_3/Listing3_27.cfm
http://localhost:8888/chapter_3/Listing3_27.cfm
Google

```
Struct
FirstName   string Steven
LastName    string Smith
or
Struct
CarType     string Audi
HouseType   string Bungalow
```

This now displays the keys maintaining their case.

What just happened?

Using Structures in Railo Server is a handy way of storing multiple values. They are stored in alphabetical order by the key name and by default, the keys are stored in uppercase, regardless of their original name.

Structures are very handy for storing simple and complex objects that you need to reference by name. The problem arises when you need to store data in an ordered manner; this is where Array variables come in.

Array variables

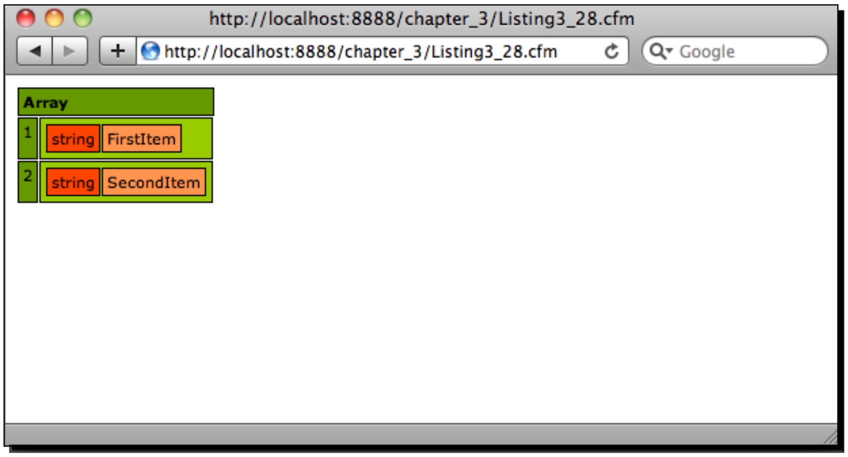
Arrays are a type of variable that can contain other variables in an indexed order. You can add any type of variable to an array, and they will be stored in the order that they were added.

Time for action – creating an array

Let's look an example of creating an array, create a file in the `Chapter_3` directory we have been using called `Listing3_28.cfm` with the following content:

```
<cfset myArray = ArrayNew(1)>
<cfset ArrayAppend(myArray, "FirstItem")>
<cfset myArray[2] = "SecondItem">
<cfdump var="#myArray#">
```

When we run this code we get the following:



http://localhost:8888/chapter_3/Listing3_28.cfm
http://localhost:8888/chapter_3/Listing3_28.cfm
Google

```
Array
1 string FirstItem
2 string SecondItem
```

What just happened?

In the previous code, we created an array using the `ArrayNew(1)` function, and passing in the variable 1 to create a one-dimensional array. Then, we used the function `ArrayAppend()` to add the string "First Item" to the array. Then, instead of using `ArrayAppend()` again, we just defined the position of the array that we wanted to add another item into. To access a variable inside an array, it is simply a matter of defining which item you want using the square brackets notation, **[]**:

```
<cfset myArray = ArrayNew(1)>
<cfset ArrayAppend(myArray, "FirstItem")>
<cfset ArrayAppend(myArray, "Second Item")>
<cfoutput>
  1st Item : #myArray[1]#
  <br>
  2nd Item: #myArray[2]#

</cfoutput>
```

The above code would output:

1st Item : First Item

2nd Item: Second Item

You can loop through the variables in an array using the `<cfloop>` tag:

```
<cfloop array="#myArray#" index="item">
  <cfoutput>
    #item#<br>
  </cfoutput>
</cfloop>
```

This would give us the following:

First Item

Second Item

In the loop above, we don't need to pass the item to the `myArray` variable, it already returns the item in the array that we are seeking in the item variable.

CFML scopes

Railo Server provides special structures that are available at different points in the request lifecycle. These structures are called Scopes and they essentially store the variables. You are able to read and write (to most of them) to help with the efficiency of your application. These scopes have names that are reserved in Railo CFML, so you can always make sure you can access them.

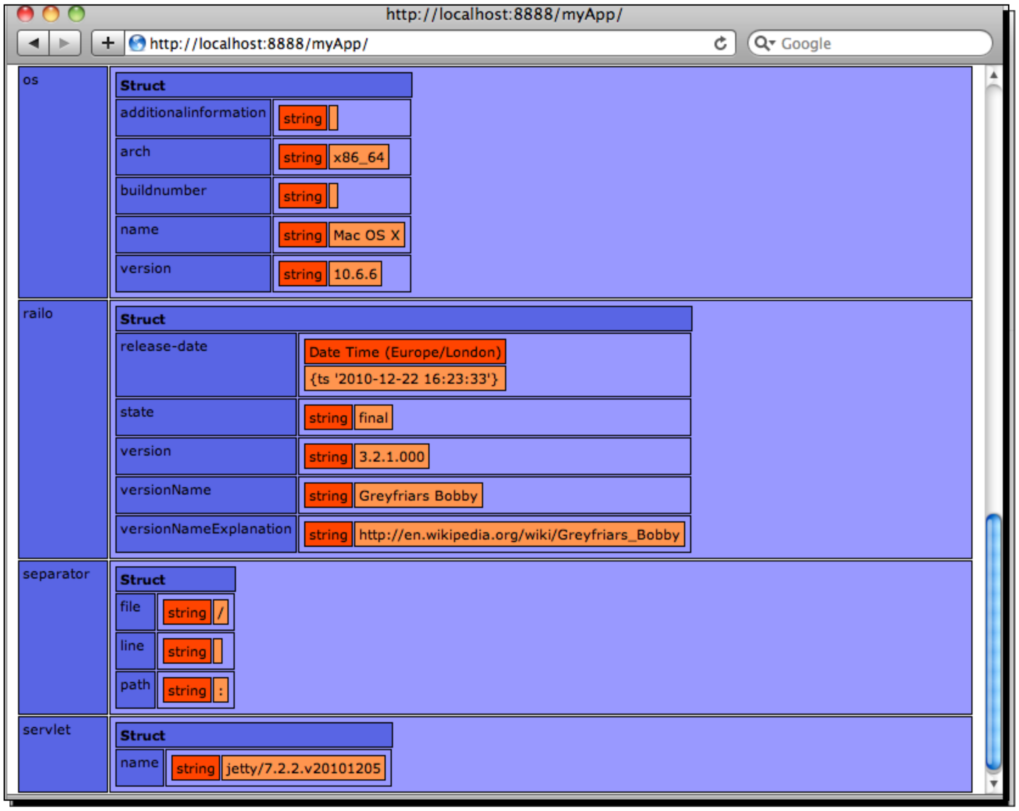
Let's have a look at some of these scopes and get an idea on how they work.

SERVER scope

The Server Scope in Railo Server is a scope that is available to all your applications on a server. Hence the name. It also provides some interesting information that you are able to read. If we want to see what the Server scope contains, all you have to do is use the `<cfdump>` tag to display the values:

```
<cfdump var="#SERVER#">
```

This displays the scope as follows:



```
os          Struct
              additionalinformation   string
              arch                    string  x86_64
              buildnumber             string
              name                    string  Mac OS X
              version                 string  10.6.6
railo       Struct
              release-date            Date Time (Europe/London)
                                      {ts '2010-12-22 16:23:33'}
              state                   string  final
              version                 string  3.2.1.000
              versionName             string  Greyfriars Bobby
              versionNameExplanation  string  http://en.wikipedia.org/wiki/Greyfriars_Bobby
separator   Struct
              file   string  /
              line   string
              path   string  :
servlet     Struct
              name   string  jetty/7.2.2.v20101205
```

This scope holds information about Railo's version and compatibility, Java Memory and version information, Operating System details, System delimiters, and which servlet container we are running.

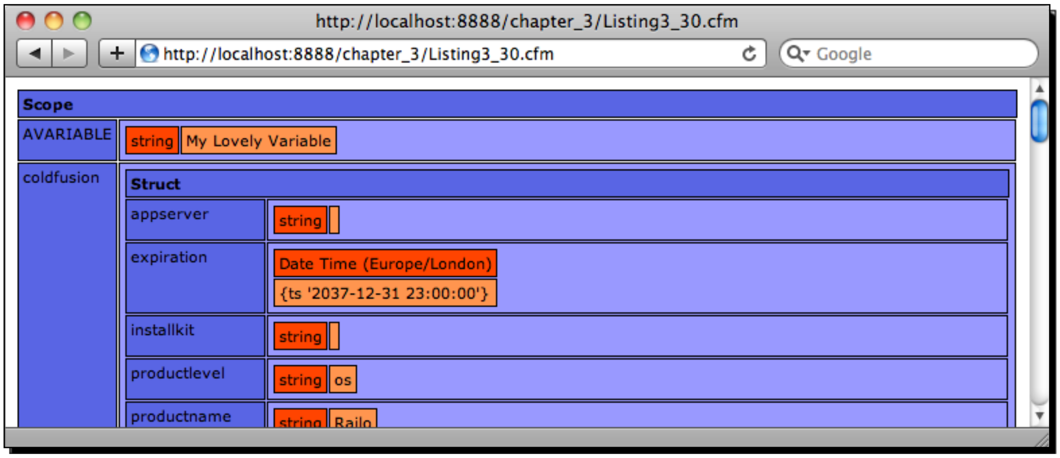
If you want to set some information in this scope, all you have to do is use the `<cfset>` tag and use the `SERVER` prefix in the variable name and set whichever value you want, to assign to it; this goes for all the other scopes too.

Time for action – adding a variable to the SERVER scope

Let's add the following code to a template called `Listing3_30.cfm` in our `Chapter_3` directory:

```
<cfset SERVER.aVariable = "My Lovely Variable">
```

Now, if you dump the `SERVER` scope you should get your variable displayed:


http://localhost:8888/chapter_3/Listing3_30.cfm
http://localhost:8888/chapter_3/Listing3_30.cfm
Google

```
Scope
AVARIABLE    string  My Lovely Variable
coldfusion   Struct
             appserver     string
             expiration    Date Time (Europe/London)
                           {ts '2037-12-31 23:00:00'}
             installkit    string
             productlevel  string  os
             productname   string  Railo
```

What just happened?

The `SERVER` scope is accessible by all the applications and contexts that are on a single instance of Railo Server. This means that if we set any variables here, they can be read (and written) across the server.

APPLICATION scope

In Railo Server, you can define a folder (and of course, its sub folders) as a specific application, with its own settings and data sources using CFML. To do this, you need to put a special file in the root named `Application.cfc`. This file has a special type of template called Component (we shall have a more detailed look at the components later in this chapter).

Time for action – creating the APPLICATION scope

Why don't we try that now:

1. In the `<railo install directory>/webroot/` folder, create another folder called `myApp`
2. In the `myApp` folder create a file called `Application.cfc`.
3. Edit `Application.cfc` and put the following code in there:

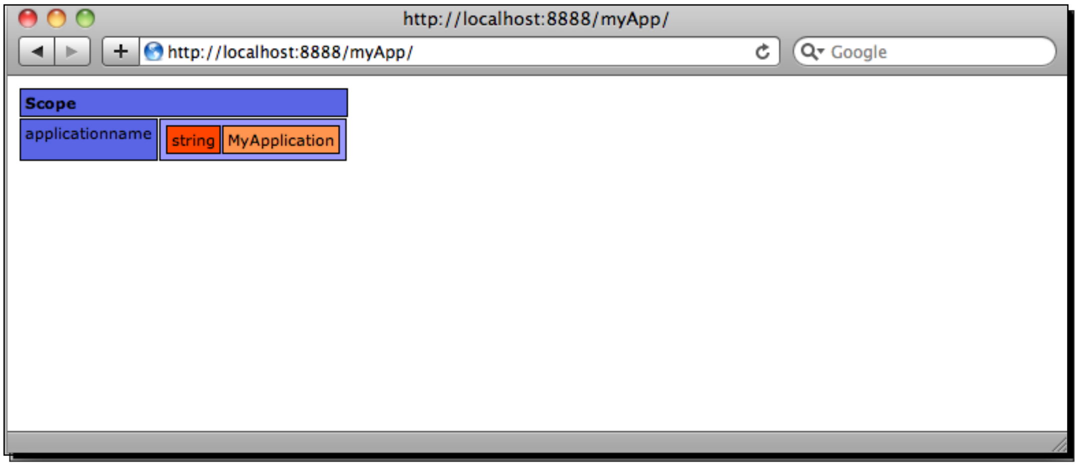
```
<cfcomponent>
  <cfset this.name = "MyApplication">
</cfcomponent>
```

4. Save the file and now create another file called `index.cfm`

5. The `index.cfm` file is the actual file we are going to call, so let's put some code in there to make sure we are in a CFML application. Add the following code to your `index.cfm`:

```
<cfdump var="#APPLICATION#">
```

6. You will see the application scope displayed, with a key called `applicationname`:



Scope	
applicationname	string MyApplication

What just happened?

By simply placing a file named `Application.cfc` and putting some settings in there we have created a scope that is available only to the templates that are in the same folder or folders below it. This way we can make sure all the settings are inherited and data can be shared only in that location.

There is much more to the application scope and application lifecycle, which we will cover in *Chapter 5*, but for now you can see that these settings are specific to your application.

SESSION scope

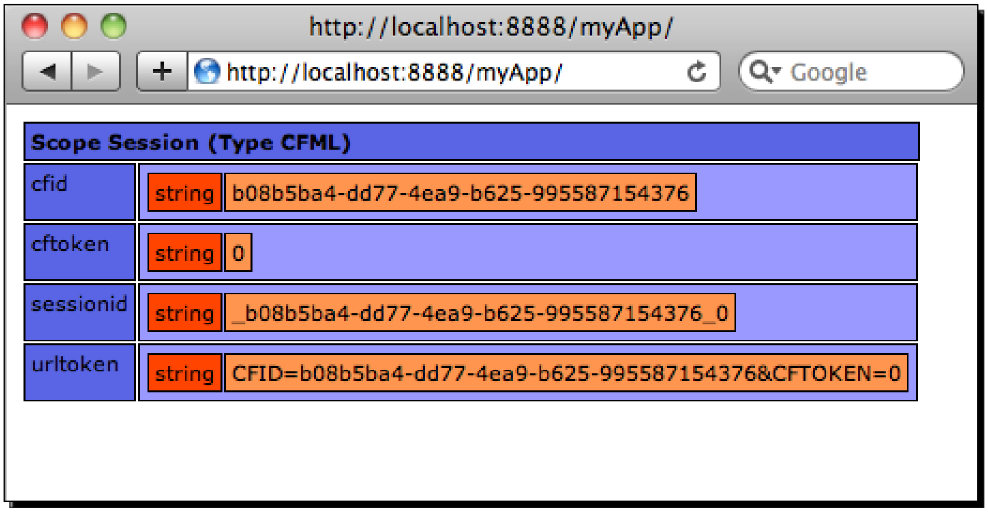
The `SESSION` scope is a structure in which you can store information about a single user across requests. This is useful to store user specific information, such as their ID. What is great about this scope is that it will survive requests, so you don't need to worry about re-setting variables. A better way to see this is to have a go yourself.

Time for action – creating a SESSION scope in your Application

Still within our myApp folder, edit the Application.cfc file and let's turn on session management—this tells our application that we want to enable sessions per user. We do this by adding <cfset this.sessionmanagement = true> in our code:

```
<cfcomponent>
  <cfset this.name = "MyApplication">
  <cfset this.sessionmanagement = true>
</cfcomponent>
```

1. If you now change the code in index.cfm to <cfdump var="#SESSION#"> and call the page, you will see the following:

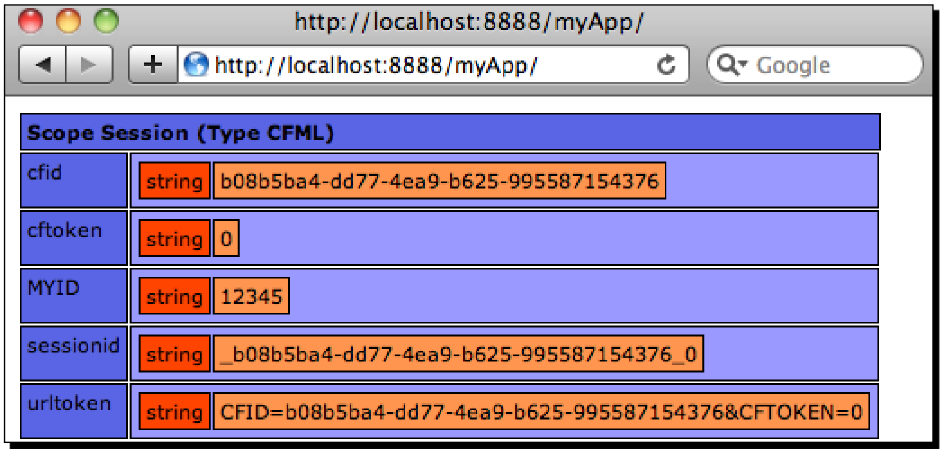


Scope Session (Type CFML)		
cfid	string	b08b5ba4-dd77-4ea9-b625-995587154376
cftoken	string	0
sessionid	string	_b08b5ba4-dd77-4ea9-b625-995587154376_0
urltoken	string	CFID=b08b5ba4-dd77-4ea9-b625-995587154376&CFTOKEN=0

2. Let's set a variable, so let's change our index.cfm to the following:

```
<cfset SESSION.myID = "12345">
<cfdump var="#SESSION#">
```

3. By running the code we now see that our variable `myID` is in the session:



Scope Session (Type CFML)

cfid	string	b08b5ba4-dd77-4ea9-b625-995587154376
cftoken	string	0
MYID	string	12345
sessionid	string	_b08b5ba4-dd77-4ea9-b625-995587154376_0
urltoken	string	CFID=b08b5ba4-dd77-4ea9-b625-995587154376&CFTOKEN=0

4. Now that that variable has been set, let's remove the code `<cfset SESSION.myId = "12345">` from `index.cfm` so that `myID` will not be set in the session scope again. When you reload the page, you'll still find our variable.

What just happened?

The `SESSION` scope will be unique for each user, hence it's very useful to store data that is specific to each user of your application there, for example, login information.

This is the power of sessions, the variables we set there maintain even across a user request.

However, sessions do not live forever. They will time out after a period of time. You can set the length of the `SESSION` variable by putting the `this.sessiontimeout` variable in your `Application.cfc` and using the `CreateTimeSpan()` function to define how long you want it to last.

REQUEST scope

The `REQUEST` scope is a structure that stores variables for the lifecycle of a single request for a page. Even if you include other files into your main file, you will still see the variables stored in the `REQUEST` scope. Like the `SERVER` and `APPLICATION` scopes, you can set variables to this scope and know they will exist for *that* user and to *that* request.

Time for action – using the REQUEST Scope

To demonstrate how this works, let's see what is in the request, include another file, set variables there and see what we end up with. Let's get started:

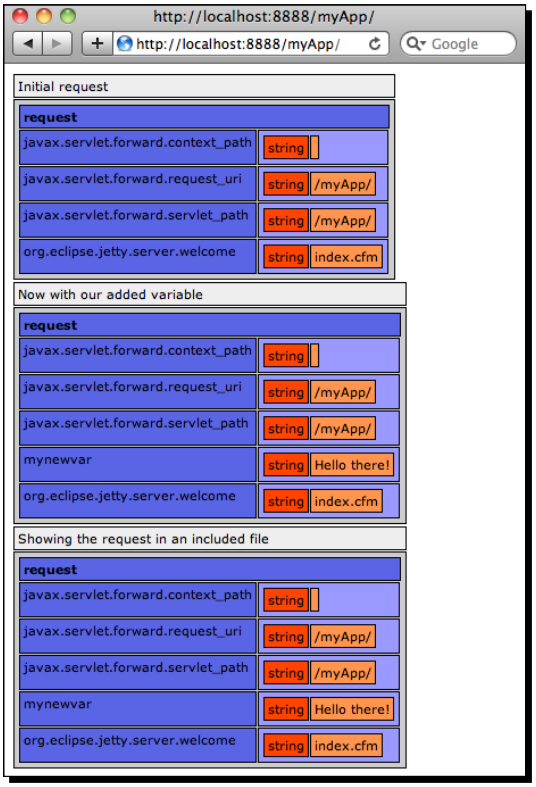
1. In the index.cfm file you created, put the following code:

```
<cfdump var="#REQUEST#" label="Initial request">
<cfset REQUEST.myNewVar = "Hello there!">
<cfdump var="#REQUEST#" label="Now with our added variable">
<cfinclude template="included.cfm">
```

2. Now, let's create another file in which we'll include the `<cfinclude>` tag. We'll name it as `included.cfm` and put the following code inside it:

```
<cfdump var="#REQUEST#" label="Showing the request scope in an
  included file">
```

3. Save the file and now run it in the web browser by going to `http://localhost:8888/myApp`; we should see the following:



```
Initial request
request
javax.servlet.forward.context_path     string
javax.servlet.forward.request_uri      string  /myApp/
javax.servlet.forward.servlet_path     string  /myApp/
org.eclipse.jetty.server.welcome       string  index.cfm

Now with our added variable
request
javax.servlet.forward.context_path     string
javax.servlet.forward.request_uri      string  /myApp/
javax.servlet.forward.servlet_path     string  /myApp/
mynewvar                               string  Hello there!
org.eclipse.jetty.server.welcome       string  index.cfm

Showing the request in an included file
request
javax.servlet.forward.context_path     string
javax.servlet.forward.request_uri      string  /myApp/
javax.servlet.forward.servlet_path     string  /myApp/
mynewvar                               string  Hello there!
org.eclipse.jetty.server.welcome       string  index.cfm
```

What just happened?

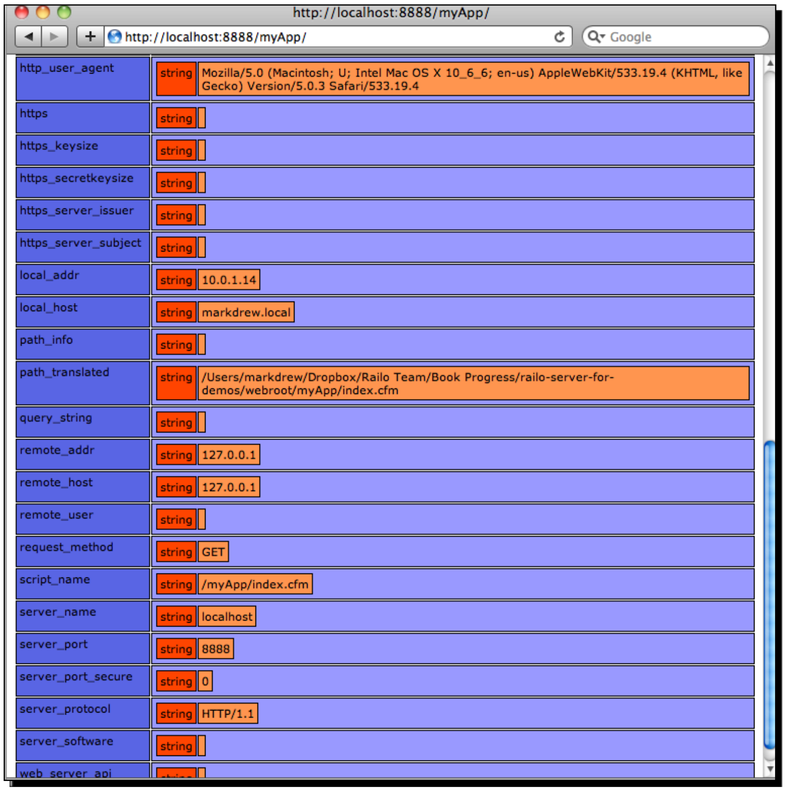
As you can see, the value that we added in `index.cfm` is available in the request scope of the included file.

The `REQUEST` scope is available to ALL the templates. It doesn't matter how they are included for the lifetime of a single request to Railo Server.

> As you have seen, `<cfdump>` is a very handy tag to see the values of complex variables, but we can also add a title to the outputted display through the `label` attribute.

CGI scope

The CGI scope is a read only structure that Railo Server provides for you to find out information about the web server and request that has been passed to you, it gives you the information about the browser and server including the client's IP address amongst other useful items. You can have a look at the contents by doing our usual `<cfdump var="#CGI#">`, which will give you something similar to the following:



```
http_user_agent       string  Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_6; en-us) AppleWebKit/533.19.4 (KHTML, like
                              Gecko) Version/5.0.3 Safari/533.19.4
https                 string
https_keysize         string
https_secretkeysize   string
https_server_issuer   string
https_server_subject  string
local_addr            string  10.0.1.14
local_host            string  markdrew.local
path_info             string
path_translated       string  /Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-
                              demos/webroot/myApp/index.cfm
query_string          string
remote_addr           string  127.0.0.1
remote_host           string  127.0.0.1
remote_user           string
request_method        string  GET
script_name           string  /myApp/index.cfm
server_name           string  localhost
server_port           string  8888
server_port_secure    string  0
server_protocol       string  HTTP/1.1
server_software       string
web_server_api        string
```

There are other scopes that are very useful in developing web applications, and we shall have a look at them in the next section.

Handling web data

So far we have been looking at the general scopes that are available to get information from the server and environment, in addition to setting variables to these scopes. In this section, we are going to look at how to handle user input, that is requests made through the URL and through web forms, which are the most common ways that users interact with your web application.

URL variables

In many sites you would have seen something like: `http://www.somesite.com/getproduct.cfm?productid=1232`. These query strings allow the server to know that you are looking for a product with the `productid` of `1232`. But how to get this information with Railo Server? Well, it's rather easy, we have the URL scope.

Time for action – getting variables from the URL

Let's see how we can get some information from the user, shall we?

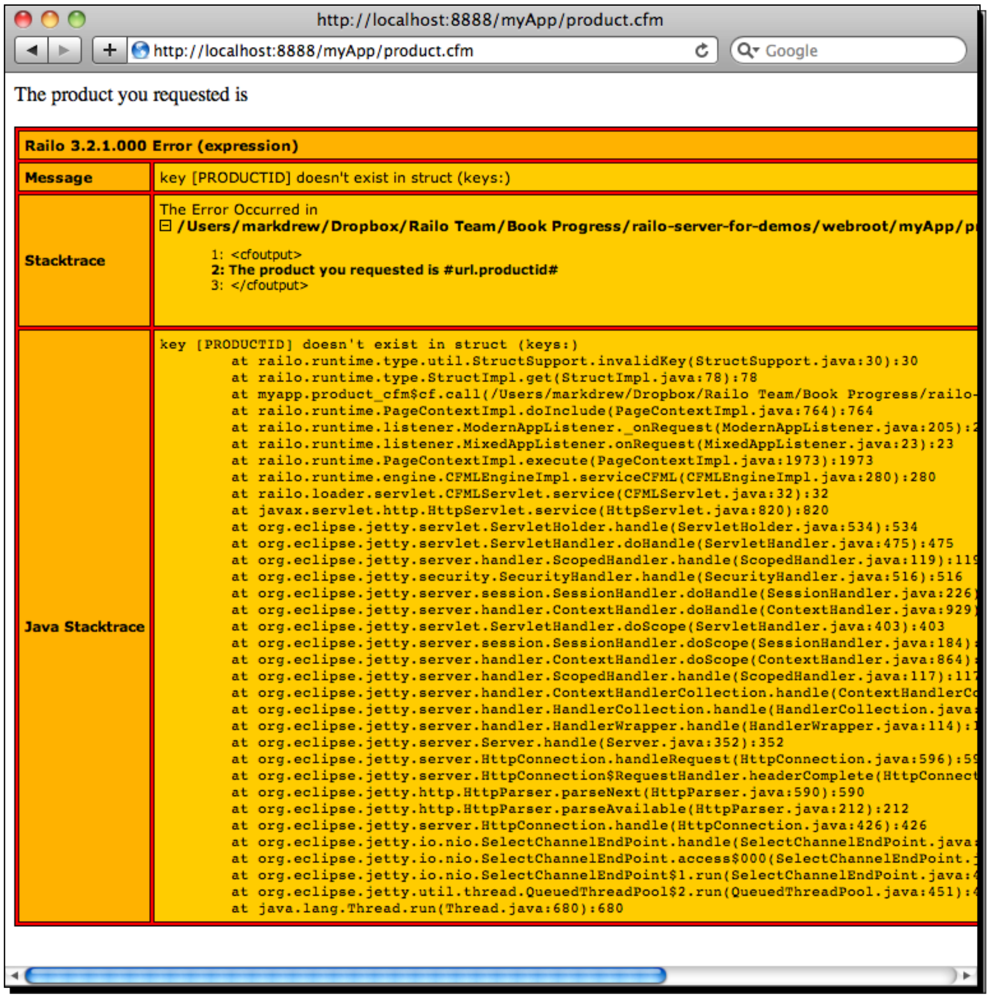
1. Create a file named `product.cfm` in your `<Railo Install Directory>/webroot/myApp/` folder.
2. Run that file in your browser with the following URL: `http://localhost:8888/myApp/product.cfm`, and you will get a blank page.
3. Now let's add the following code:

```
<cfoutput>
The product you requested is #url.productid#
</cfoutput>
```

Now let's browse to the page with the following: `http://localhost:8888/myApp/product.cfm?productid=1234`. You will now see "The product you requested is 1234" displayed on the page.

It is that simple to access variables passed in the URL; all you have to do is to reference the URL scope.

But what happens if we remove the `?productid=1234` from the URL? Oh dear! We get an error!


http://localhost:8888/myApp/product.cfm
http://localhost:8888/myApp/product.cfm
Google

The product you requested is

Railo 3.2.1.000 Error (expression)

Message	key [PRODUCTID] doesn't exist in struct (keys:)
Stacktrace	The Error Occurred in ⊟ **/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/myApp/p** 1: <cfoutput> **2: The product you requested is #url.productid#** 3: </cfoutput>

Java Stacktrace

```
key [PRODUCTID] doesn't exist in struct (keys:)
        at railo.runtime.type.util.StructSupport.invalidKey(StructSupport.java:30):30
        at railo.runtime.type.StructImpl.get(StructImpl.java:78):78
        at myapp.product_cfm$cf.call(/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-
        at railo.runtime.PageContextImpl.doInclude(PageContextImpl.java:764):764
        at railo.runtime.listener.ModernAppListener._onRequest(ModernAppListener.java:205):2
        at railo.runtime.listener.MixedAppListener.onRequest(MixedAppListener.java:23):23
        at railo.runtime.PageContextImpl.execute(PageContextImpl.java:1973):1973
        at railo.runtime.engine.CFMLEngineImpl.serviceCFML(CFMLEngineImpl.java:280):280
        at railo.loader.servlet.CFMLServlet.service(CFMLServlet.java:32):32
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:820):820
        at org.eclipse.jetty.servlet.ServletHolder.handle(ServletHolder.java:534):534
        at org.eclipse.jetty.servlet.ServletHandler.doHandle(ServletHandler.java:475):475
        at org.eclipse.jetty.server.handler.ScopedHandler.handle(ScopedHandler.java:119):119
        at org.eclipse.jetty.security.SecurityHandler.handle(SecurityHandler.java:516):516
        at org.eclipse.jetty.server.session.SessionHandler.doHandle(SessionHandler.java:226)
        at org.eclipse.jetty.server.handler.ContextHandler.doHandle(ContextHandler.java:929)
        at org.eclipse.jetty.servlet.ServletHandler.doScope(ServletHandler.java:403):403
        at org.eclipse.jetty.server.session.SessionHandler.doScope(SessionHandler.java:184):
        at org.eclipse.jetty.server.handler.ContextHandler.doScope(ContextHandler.java:864):
        at org.eclipse.jetty.server.handler.ScopedHandler.handle(ScopedHandler.java:117):117
        at org.eclipse.jetty.server.handler.ContextHandlerCollection.handle(ContextHandlerCo
        at org.eclipse.jetty.server.handler.HandlerCollection.handle(HandlerCollection.java:
        at org.eclipse.jetty.server.handler.HandlerWrapper.handle(HandlerWrapper.java:114):1
        at org.eclipse.jetty.server.Server.handle(Server.java:352):352
        at org.eclipse.jetty.server.HttpConnection.handleRequest(HttpConnection.java:596):59
        at org.eclipse.jetty.server.HttpConnection$RequestHandler.headerComplete(HttpConnect
        at org.eclipse.jetty.http.HttpParser.parseNext(HttpParser.java:590):590
        at org.eclipse.jetty.http.HttpParser.parseAvailable(HttpParser.java:212):212
        at org.eclipse.jetty.server.HttpConnection.handle(HttpConnection.java:426):426
        at org.eclipse.jetty.io.nio.SelectChannelEndPoint.handle(SelectChannelEndPoint.java:
        at org.eclipse.jetty.io.nio.SelectChannelEndPoint.access$000(SelectChannelEndPoint.j
        at org.eclipse.jetty.io.nio.SelectChannelEndPoint$1.run(SelectChannelEndPoint.java:4
        at org.eclipse.jetty.util.thread.QueuedThreadPool$2.run(QueuedThreadPool.java:451):4
        at java.lang.Thread.run(Thread.java:680):680
```

Our applications should be a bit more robust, right? Thankfully, we can easily fix that with a simple tag. Let's update our code here:

```
<cfparam name="URL.productid" default="">
<cfoutput>
The product you requested is #url.productid#
</cfoutput>
```

We can now load our page without a single error.

What just happened?

Accessing the variables that have been passed in the URL scope is rather easy, because it is just another structure as we have used before. What we have to make sure is that our applications are a bit more robust and variables are well defined if we are going to use them. One option is to use the `<cfparam>` tag. This tag sets a default variable to the `URL.productid` to a blank string if it isn't passed in. This is a good way to make sure your code is robust and can handle unforeseen user actions.

Other ways to check if the variables are present is to use the `isDefined("URL.productid")` function, passing in the variable (in quotes) that you want to check, or to explicitly check if the variable exists in a structure with the function `StructKeyExists(URL, "productid")`

Have a Go Hero – try the isDefined() and StructKeyExists() functions

Why not try re-writing the code in our template using the `isDefined()` and `StructKeyExists()` functions. It should be easier by now!

FORM variables

In many web applications, we need to get information from the user using a form, be it a feedback form, a contact us form, a registration form, or a search dialog. Railo Server has some handy ways to let you access this information in the `FORM` scope. This scope is created when a form has been posted to another template using the `POST` method in the form.

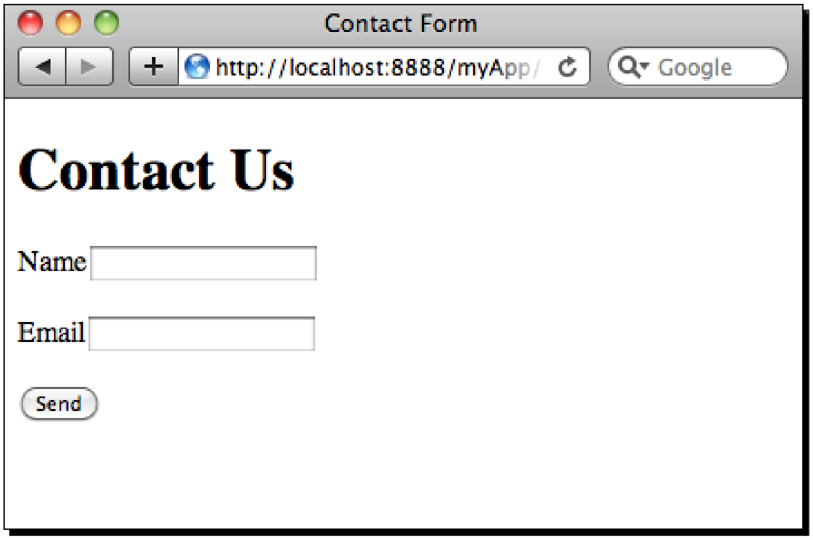
Time for action – getting FORM variables

Let's look at a simple form and check out what gets sent:

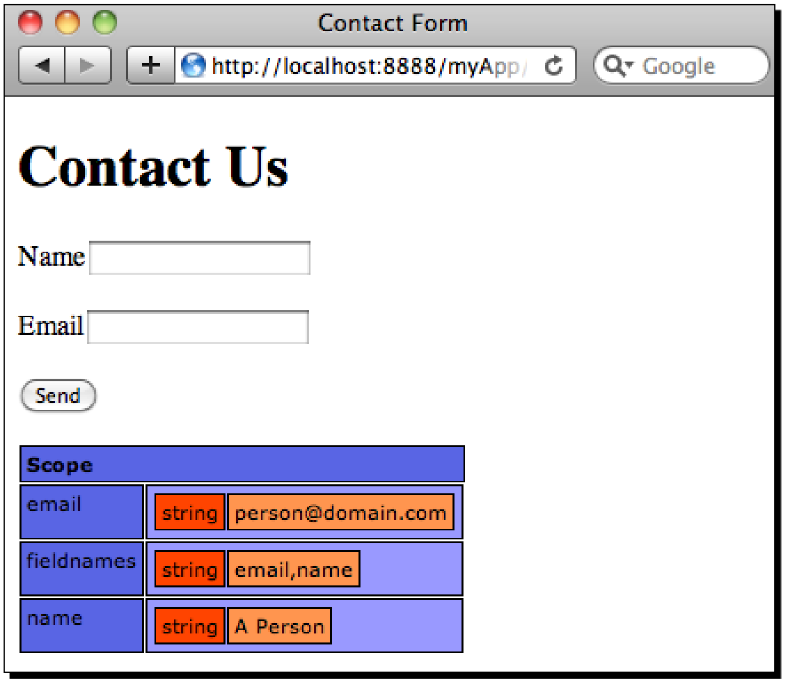
1. Create a file in your application by the name `contact.cfm` and create a simple HTML page within this file:

```
<!DOCTYPE html >
<html lang="en">
<head>
  <title>Contact Form</title>
</head>
<body>
<h1>Contact Us</h1>
<form action="contact.cfm" method="post">
  <p>
    <label for="name">Name</label><input type="text" name="name">
  </p>
  <p>
    <label for="email">Email</label><input type="email"
name="email" >

  </p>
  <p><input type="submit" value="Send"></p>
</form>

</body>
</html>
```

Now, if you load this template in your browser, you should see the following:



2. This form posts to itself, as you can see through `action="contact.cfm"` in the form tag. We also have `method="post"` to make sure we are accepting a form post. Now if we submit this, we will get no output because we haven't added any CFML code. Let's add a simple `<cfdump var="#FORM#">` after the form, and resubmit the form:



Scope	
email	string person@domain.com
fieldnames	string email,name
name	string A Person

3. As with the URL scope, you can reference the values directly, but if they are not present you will get an error, so let's change our code a little bit to include some checks:

```
<!DOCTYPE html >

<cfparam name="FORM.name" default="">
<cfparam name="FORM.email" default="">
<html lang="en">
<head>
  <title>Contact Form</title>
</head>
<body>
<h1>Contact Us</h1>
<form action="contact.cfm" method="post">
  <p>
    <label for="name">Name</label><input type="text" name="name">
  </p>

  <p>
    <label for="email">Email</label><input type="email"
name="email" >
  </p>
  <p><input type="submit" value="Send"></p>
</form>
```

```
<cfif Len(FORM.name) AND Len(FORM.email)>
  <cfoutput>
    Hello #FORM.name#, thanks for giving the email address #FORM.
email#

  </cfoutput>
</cfif>
</body>
</html>
```

What just happened?

Using the FORM variable is exactly same as using the URL variable, apart from the fact that it is generated only when you submit a form using the POST method.

In the previous code, we added the <cfparam> tags at the top to make sure we have a default for the **name** and **email** fields. Also, in the form, we have added a check to see if there is a length to the FORM.name variable by using the Len() function and also checking the length to the FORM.email variable.

You might be wondering about the Len(FORM.name) statement used in the previous code. If the length of the string is equal to zero, then the statement will be "false", but if it has some length, this would equate to "true."

Using the <cfparam> is recommended in any template that will be retrieving content, as you can set the default content without having to check the existence of every field you need to use.

Cookies

Cookies are a technology used by web browsers that allows web developers to store a small amounts of information for a period of time even if the browser closes. So for example you can store a user's preferences with regards to their choice of background color, or even their unique login name (read more about cookies here: http://en.wikipedia.org/wiki/HTTP_cookie).

Railo Server makes it very easy to read and save information to the browser's cookies. There is another scope available to you to read the cookies, aptly named COOKIE.

You can save a variable name and value to the COOKIE scope using the <cfset COOKIE.variableName = variableValue> but this does not give you enough control over the other attributes that you set with a cookie, such as, when it expires, which domains can access it, and maybe the path in a domain that the cookie can access. For this, we have the <cfcookie> tag.

The `<cfcookie>` tag is very simple to use, if you want to save a variable in a cookie all you have to do is:

```
<cfcookie name="superSecretName" value="Elvis">
```

Of course, we might want to say how long this cookie can live, so we can also add the `expires` attribute:

```
<cfcookie name="superSecretName" value="Elvis" expires="30/12/2011">
```

You can also set which domain the cookie can only be read from:

```
<cfcookie name="superSecretName" value="Elvis" domain="www.mydomain.
com">
```

If your application only exists within a path of the specified domain, you could also add the path:

```
<cfcookie name="superSecretName" value="Elvis" domain="www.mydomain.
com" path="/myApp">
```

To read the value from the cookie all you have to do is:

```
<cfset cookieValue = COOKIE.superSecretName>
<cfdump var="#cookieValue#">
```

Database access

So far we have looked at all the ways that we can use to display variables in Railo Server. But one of the main attractions of using Railo Server is the really easy way you can access databases.

Railo Server makes it really easy to define and query data stored in any **Relational Database Management System** (**RDBMS**) virtually. In this section, we shall go through the setup of a data source to a database, running queries against that database, securing our queries against SQL Injection attacks and even running stored procedures.

One of the main functions that are used in nearly every web application is to show and/or capture data from its users. Railo Server makes this incredibly simple with minimal code so that we can see how these things work together; of course, we need to have a database to connect to.

Railo Server can connect to nearly every single database out there. Out of the box (or rather out of the ZIP) Railo Server can connect to DB2, Firebird, HSSQl (Hipersonic), Microsoft MS SQL, MySQL, Oracle, PostgreSQL, Sybase, and any other database that has Java JDBC driver. Railo Server even includes a JDBC-ODBC bridge to connect (on Windows) to your ODBC datasources.

For the examples in this book, we chose MySQL to connect to, because it's a free and open source database that is easy to install and configure.

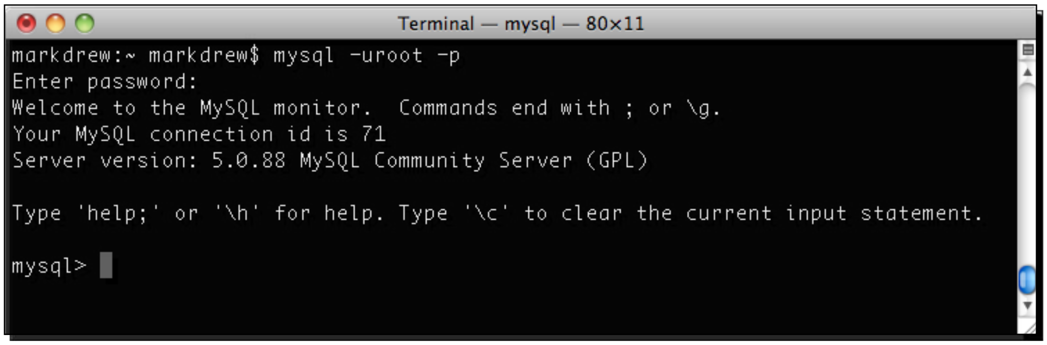
Time for action – installing MySQL and setting up our database

Before we can configure a datasource to hook onto, we might want to have a database first. For the following examples we are going to be using MySQL, the world's most popular Open Source database. You can download a version to match your operating system, for free, from: `http://dev.mysql.com/downloads/`. For the following examples, we will be using MySQL Community Server.

Apart from the database server, you can also get some tools to interact with the database visually, so once you have the database downloaded and you're running (instructions for each operating system is available on the website) MySQL, you can go ahead and create a database called "railobook". For example, if you are using the command line, it is easy to connect to MySQL with the following command:

```
$ mysql -uroot -p
```

Once you enter your password, you should have a response from the server as follows:



```
markdrew:~ markdrew$ mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 71
Server version: 5.0.88 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```

Now that we have connected we are going to go in and create a database to store our tables, use that database, create a sample table, and finally add a bit of content to it. Let's get started!

Now that we have logged in, let's go and create the database:

```
> CREATE DATABASE railobook;
```

To use the database, enter the following in the console:

```
> USE railobook;
```

Now, let's create an `employee` table; this is the table that we will be getting data by running the following code:

```
> CREATE TABLE employee (
  id int(11) NOT NULL auto_increment,
  FirstName varchar(50) default NULL,
  LastName varchar(50) default NULL,
  email varchar(100) default NULL, PRIMARY KEY  (id) );
```

The server will respond with:

Query OK, 0 rows affected (0.01 sec)

This is to tell you that it has created the table successfully.

Let's add some test data:

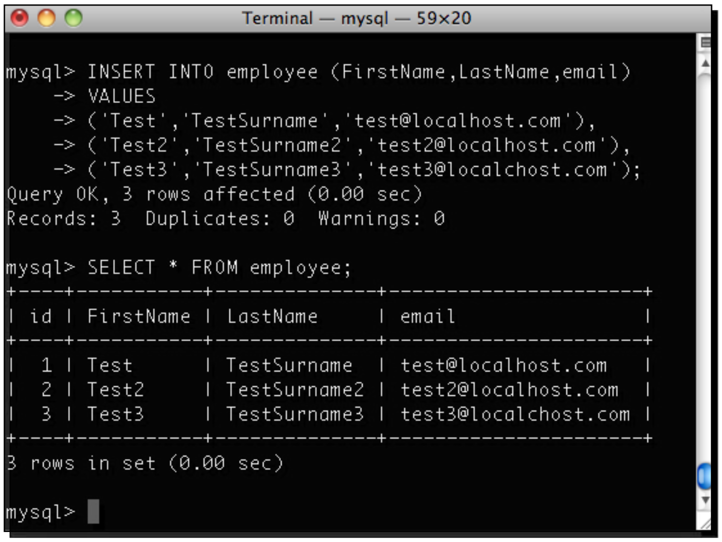
```
> INSERT INTO employee (FirstName,LastName,email)
  VALUES
  ('Test','TestSurname','test@localhost.com'),
  ('Test2','TestSurname2','test2@localhost.com'),
  ('Test3','TestSurname3','test3@localchost.com');
```

The server will respond with "Query OK, 3 rows affected (0.00 sec)" to tell you that you have created three new records in the table employee.

Finally, let's see what is stored in the `employee` table:

```
> SELECT * FROM employee;
```

You should get the results as follows:



```
mysql> INSERT INTO employee (FirstName,LastName,email)
    -> VALUES
    -> ('Test','TestSurname','test@localhost.com'),
    -> ('Test2','TestSurname2','test2@localhost.com'),
    -> ('Test3','TestSurname3','test3@localchost.com');
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM employee;
+----+-----------+--------------+----------------------+
| id | FirstName | LastName     | email                |
+----+-----------+--------------+----------------------+
|  1 | Test      | TestSurname  | test@localhost.com   |
|  2 | Test2     | TestSurname2 | test2@localhost.com  |
|  3 | Test3     | TestSurname3 | test3@localchost.com |
+----+-----------+--------------+----------------------+
3 rows in set (0.00 sec)

mysql> 
```

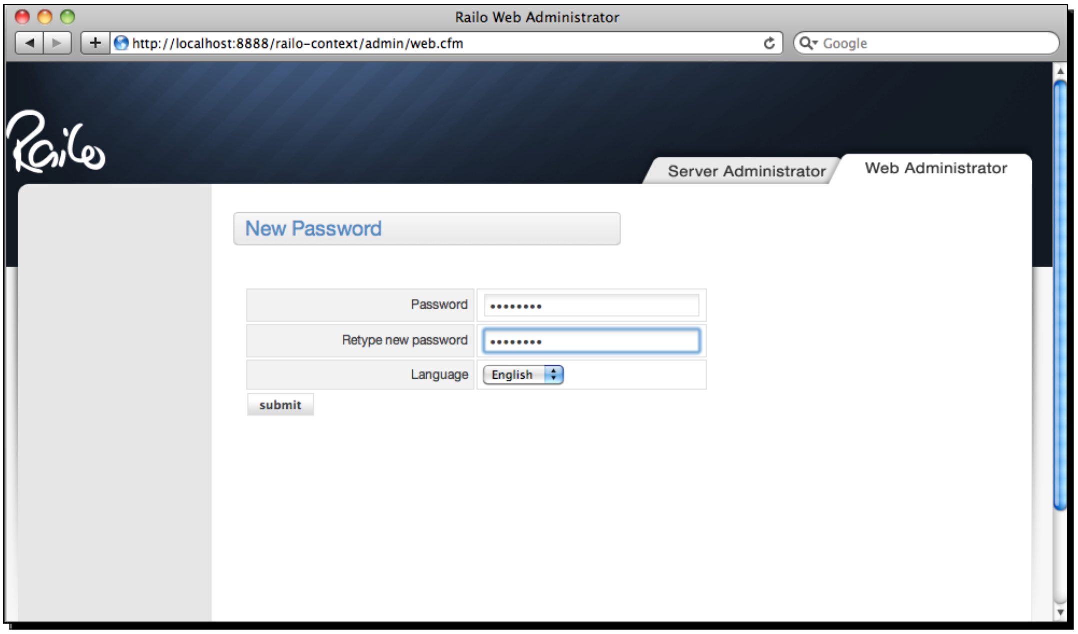
What just happened?

We installed the MySQL database server onto our machine. We then connected to the server through the command line and created a database called `railobook`. Before we can add tables and data, we switched to using the `railobook` database, then we created a table named `employees` and added some test data!

Time for action – configuring data sources in Railo Server

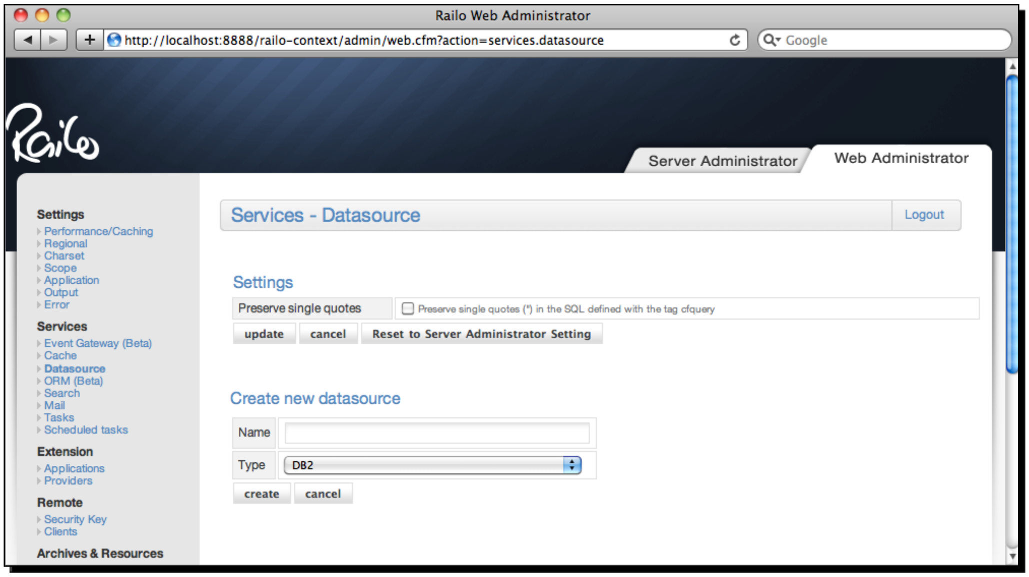
Now that we have our test data, we are going to create a data source in the Railo Server web administrator. The administrator is where you can configure many aspects of Railo Server's behavior, which we will look at (in depth) later in this chapter. Let's go and setup a datasource pointing to the `railobook` database.

In your browser, head to the administrator by going to `http://localhost:8888/railo-context/admin/web.cfm`. This takes us to the Railo Web Administrator. If you have not already done this, you can enter a password to protect this area.



On the left-hand side of the **Overview** page, there is a list of links, under the **Services** category. From this list, locate the **Datasource** link and click on it.

We can create a new datasource by entering **railobook** in the **Name** field of the **Create new datasource** form and selecting **MySQL** from the **Type** drop down. Now, press the **create** button.



Services - Datasource

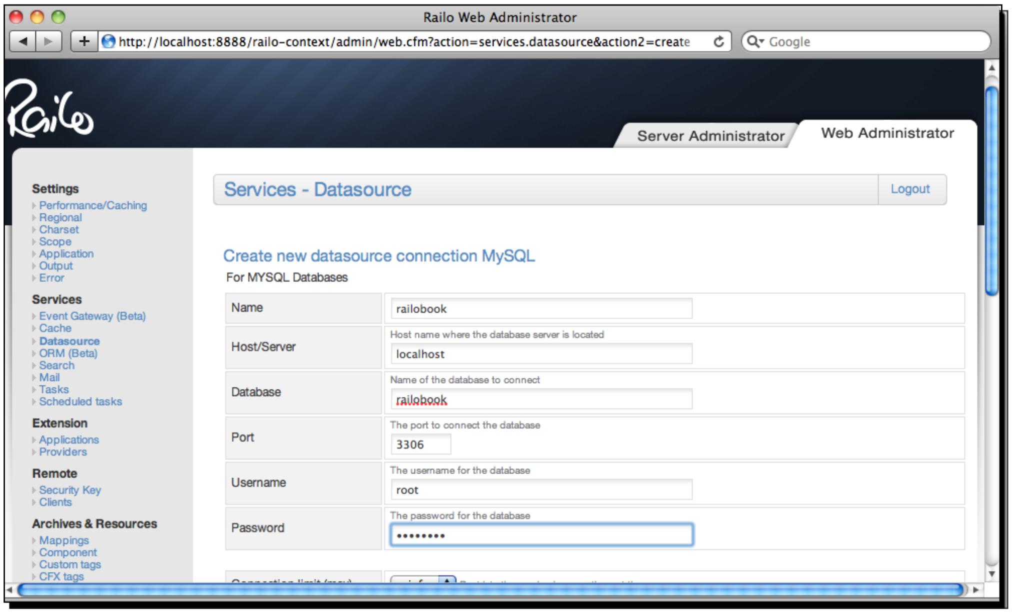
Settings

Preserve single quotes	☐ Preserve single quotes (') in the SQL defined with the tag cfquery

Create new datasource

Name	
Type	DB2

We can now set the details of the connection. In the **Database** field, we enter **railobook** as that is the name of the database that we created, we can leave the **Host/Server** and **Port** fields as they are (if you have installed MySQL on your local machine), and we can enter the username and password for this database. If you don't know the username and password for your database, just use the values that were defined during the installation of MySQL. Once done, scroll to the bottom of the form and click on **create**.



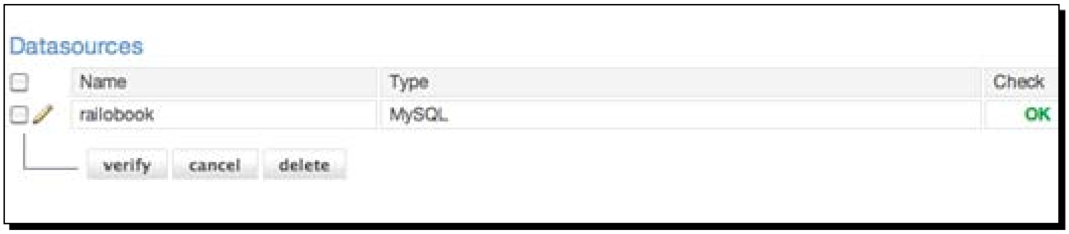
Services - Datasource

Create new datasource connection MySQL

For MYSQL Databases

Name	railobook
Host/Server	Host name where the database server is located localhost
Database	Name of the database to connect railobook
Port	The port to connect the database 3306
Username	The username for the database root
Password	The password for the database ••••••••

Congratulations! You should now have a new **Datasource** setup with a nice green **OK** verification that all went well.



Datasources

□	Name	Type	Check
□ ✎	railobook	MySQL	OK

What just happened?

The Railo Web Administrator is where connections to various systems can be defined. As we will just refer to the data source name (**railobook** in this case) in our code, it means that we don't have to change our code if we need to change the location of the database.

That is all we needed to do, now we can go and run some queries on our database!

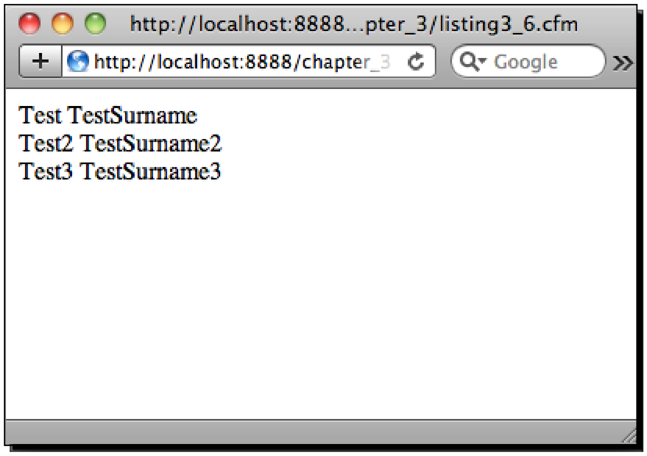
Time for action – running queries against our database

The magic for running a lot of queries with Railo stems from the `<cfquery>` tag. This is a very versatile tag that allows you to run queries to a data source. Remember, we queried the database to get our list of employees? Let's do that in CFML to see how easy it is to do.

```
<cfquery name="getEmployees" datasource="railobook">
  SELECT * FROM employee
</cfquery>
<cfloop query="getEmployees">
<cfoutput>#FirstName# #LastName# <br></cfoutput>
</cfloop>>
```

We use the `<cfquery>` tag, setting the name of the results to `getEmployees` by putting that in the `name` attribute. We also use the `datasource` attribute to say we are going to use the datasource we setup, called `railobook` previously.

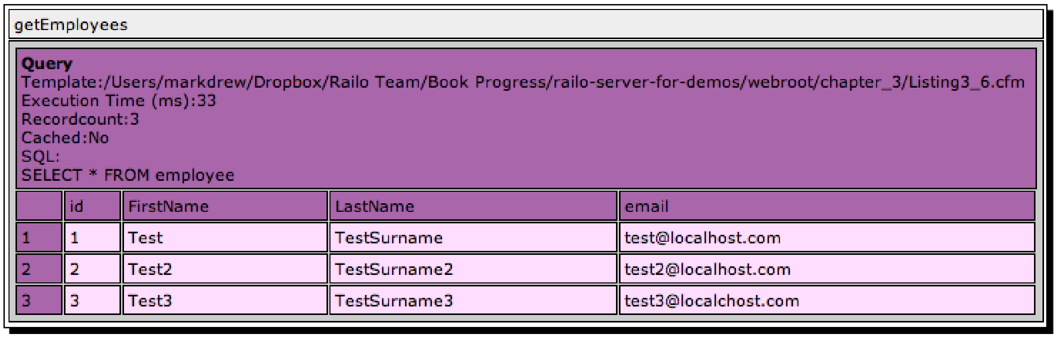
Then we put the SQL query, to select all the employees. Then to show that something is happening we use the `<cfloop>` tag and define which query you want to loop through. Then to output the variables we use the `<cfoutput>` tag and just need to put the names of the columns we want to output, surrounding the names with the # to say they should be evaluated. Running this code gets us a nice listing of our users.


http://localhost:8888...pter_3/listing3_6.cfm
http://localhost:8888/chapter_3
Google

Test TestSurname
Test2 TestSurname2
Test3 TestSurname3

There is more information you can get from a query, such as the total number of records, how long it took to execute, the template that called it, and the final rendered SQL statement. To see these details, you can use the `<cfdump>` tag as follows:

```
<cfdump eval=getEmployees>
```



getEmployees

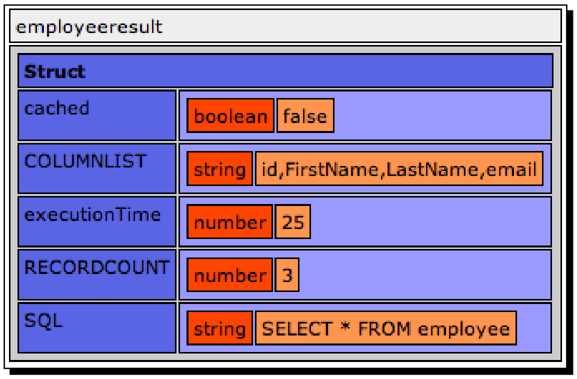
Query
Template:/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/chapter_3/Listing3_6.cfm
Execution Time (ms):33
Recordcount:3
Cached:No
SQL:
SELECT * FROM employee

	id	FirstName	LastName	email
1	1	Test	TestSurname	test@localhost.com
2	2	Test2	TestSurname2	test2@localhost.com
3	3	Test3	TestSurname3	test3@localchost.com

If you need to use these variables in your code, instead of just seeing them, you can add the `result` attribute to the `<cfquery>` tag as follows:

```
<cfquery name="getEmployees" datasource="railobook"
result="employeeresult">
  SELECT * FROM employee
</cfquery>
```

This will give you a structure that you can reference. If you use `<cfdump>`, the `employeeresult` variable you get the following:



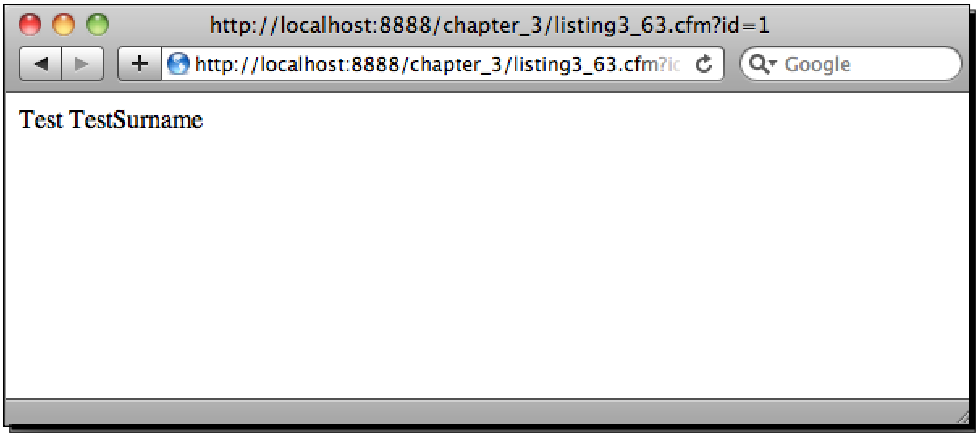
employeeresult

Struct		
cached	boolean	false
COLUMNLIST	string	id,FirstName,LastName,email
executionTime	number	25
RECORDCOUNT	number	3
SQL	string	SELECT * FROM employee

What about if we want to get one user via the `URL` (or some other variable)? This is also very easy. For example, if we needed to get a user by his/her ID (remember, we created an `id` column in the table?) from a `URL` variable, then all we need to do is to put that variable in the SQL code:

```
<cfquery name="getEmployees" datasource="railobook"
result="employeeresult">
  SELECT * FROM employee
  WHERE id = #URL.id#
</cfquery>
<cfloop query="getEmployees">
<cfoutput>#FirstName# #LastName# <br></cfoutput>
</cfloop>
```

What's left now is to open the page with "`id=1`" as one of our URL variables: `http://localhost:8888/chapter_3/listing3_63.cfm?id=1`.


http://localhost:8888/chapter_3/listing3_63.cfm?id=1
http://localhost:8888/chapter_3/listing3_63.cfm?ic
Google

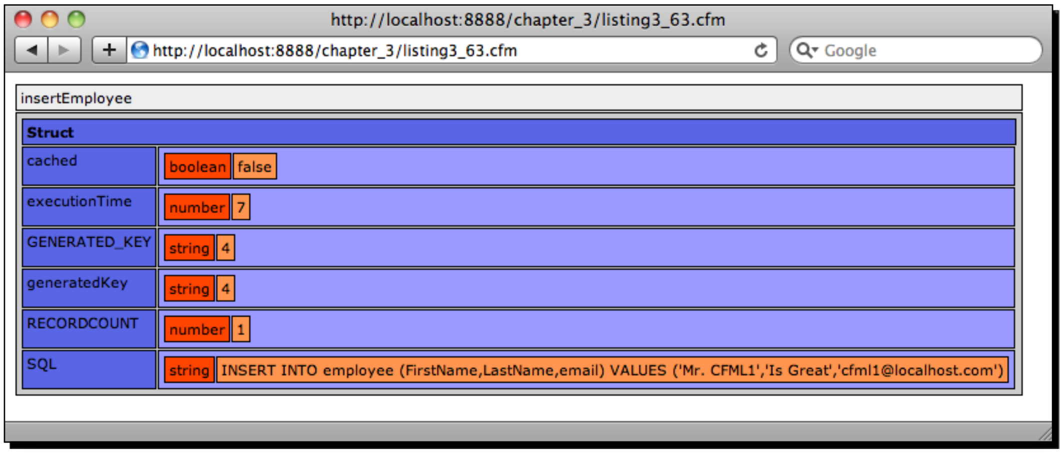
Test TestSurname

And we can see that the only record returned was the one where the ID matched the variable in the URL.

What happens if we want to insert a new record? We can just use the SQL for that in a `<cfquery>` tag.

```
<cfquery result="insertEmployee" datasource="railobook">
  INSERT INTO employee (FirstName,LastName,email)
  VALUES
    ('Mr. CFML1','Is Great','cfml1@localhost.com')

</cfquery>

<cfdump eval=insertEmployee>
```

In the above line, we have removed the `name` attribute from the `<cfquery>` tag because it won't return any results. We then add our insert SQL statement and then use `<cfdump>` to see what information was returned about our last query.


http://localhost:8888/chapter_3/listing3_63.cfm
http://localhost:8888/chapter_3/listing3_63.cfm
Google

```
insertEmployee
Struct
cached          boolean false
executionTime   number  7
GENERATED_KEY   string  4
generatedKey    string  4
RECORDCOUNT     number  1
SQL             string  INSERT INTO employee (FirstName,LastName,email) VALUES ('Mr. CFML1','Is Great','cfml1@localhost.com')
```

Queries with parameters

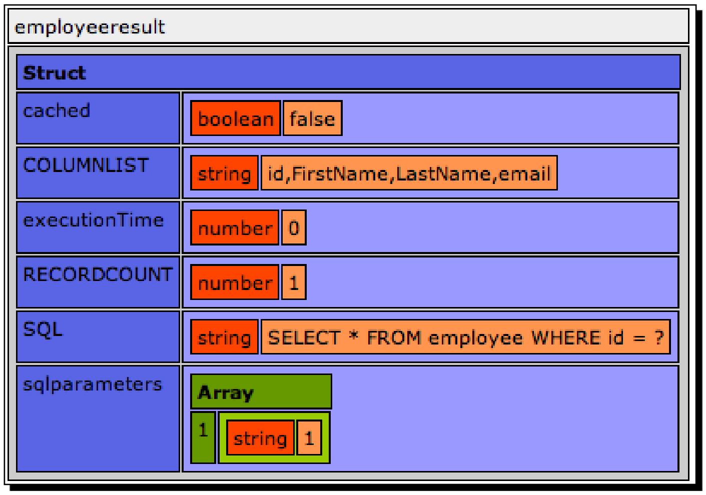
When running queries against records in a database from your web application you have to be very cautious to sanitize the data that the users pass to your application. Malicious users can use a method called SQL Injection (`http://en.wikipedia.org/wiki/SQL_injection`) to delete, add, and generally tamper with your data that may result in a compromise of security and stability of your application.

CFML has a very simple way to stop this type of attack by checking the variables that you are passing to your database with the use of `<cfqueryparam>`. This tag allows you to set the value and the type of variable that you are passing in your SQL statement and also improves the performance as the database will cache the statement without variables and return the values faster (this is called a prepared statement).

So, how can we implement our select employee query with a `<cfqueryparam>`? We simply replace our variable as so:

```
<cfquery name="getEmployees" datasource="railobook"
result="employeeresult">
  SELECT * FROM employee
  WHERE id = <cfqueryparam cfsqltype="cf_sql_numeric" value="#URL.
id#">
</cfquery>


<cfloop query="getEmployees">
<cfoutput>#FirstName# #LastName# <br></cfoutput>
</cfloop>
```

If we now check what the query information is by dumping the `employeeresult` variable we shall see that SQL has added a "?" to our user ID:



```
employeeresult
Struct
cached          boolean  false
COLUMNLIST      string   id,FirstName,LastName,email
executionTime   number   0
RECORDCOUNT     number   1
SQL             string   SELECT * FROM employee WHERE id = ?
sqlparameters   Array
                1  string  1
```

What just happened?

In all the above examples we are able to see how easy it is to work with Databases by using the `<cfquery>` tag. We can retrieve records and insert records (you could try deleting and updating records too) easily.

By using the <cfqueryparam> variable, we have seen how we can prevent SQL Injection attacks and define specifically the format of the data we are querying.

Stored procedures

Databases have special functions that you can create called Stored Procedures (http://en.wikipedia.org/wiki/Stored_procedure) that allow you to have more complex query logic stored within the database itself. If you need to call these stored procedures from Railo Server it is a simple matter of using the <cfstoredproc> tag.

Time for action – calling stored procedures

Let's create a stored procedure in our database and run it from Railo Server.

In our MySQL console, let's run the following command to create a stored procedure called employeebyid:

```
DELIMITER //
CREATE PROCEDURE employeebyid(IN empid int(11))
BEGIN
   SELECT *
  FROM employee
   WHERE id = empid;
END //
DELIMITER ;
```

We get a response from MySQL as follows:

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE employeebyid(IN empid int(11))
    -> BEGIN
    ->    SELECT *
    -> FROM employee
    ->  WHERE id = empid;
    -> END //
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
mysql>
```

Now, we can test a call to it by running the following code:

```
mysql> CALL employeebyid(1);
+----+-----------+------------+-------------------+
| id | FirstName | LastName   | email             |
+----+-----------+------------+-------------------+
|  1 | Test      | TestSurname | test@localhost.com |
+----+-----------+------------+-------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

So, now our stored procedure works from inside MySQL. Let's call it from a CFML template:

```
<cfstoredproc procedure="employeebyid" datasource="railobook">
  <cfprocparam  cfsqltype="cf_sql_numeric" value="1" />
  <cfprocresult name="getEmployee" />
</cfstoredproc>

<cfdump eval=getEmployee>.
```

We first call the `<cfstoredproc>` procedure, passing it the procedure name and the datasource. To pass a variable to the stored procedure we need to nest the `<cfprocparam>` tag, defining the data type we are passing using the `cf_sql_numeric` for a general numeric type and its value. Then, we add the `<cfprocresult>` tag, this will grab the results from the stored procedure and save them to the `getEmployee` variable that we can use later on. It is that simple!

Of course, stored procedures can get trickier with them returning multiple variables and taking in a number of parameters, like `cachedAfter` or `cacheWithin` to perform query caching, debug for listing debug information about each stored procedure statement executed and so on, but you should now have a good example of how to deal with even more complex calls.

Object Oriented Programming with Components

Object Oriented Programming (OOP, `http://en.wikipedia.org/wiki/Object-oriented_programming`) is a paradigm that has been available in a number of languages for decades and, in fact, most modern programming languages provide support for defining objects with properties and methods. This is the same with CFML.

Railo Server provides OOP capabilities to developers in the form of components. So far, you have seen CFML templates created with the file extension `.cfm`. CFML components are defined with the `.cfc` extension.

So, what are objects?

Objects in OOP allow you to encapsulate blocks of code that define a specific business object, for example, an employee. An `employee` object in an application would be a way to interact with data of an employee. It is not just properties of an employee, for example, first name, surname, salary, and e-mail, that can be encapsulated in an object, but also functions (or methods), for example, `getEmployeeName` and `setEmployeeName`, that you can call to get and set properties, or perform actions on that object.

Time for action – creating the Employee component

To define an object in CFML, you need to create a file with the extension `.cfc`, let's create the file `Employee.cfc` and put the following code inside it:

```
<cfcomponent name="Employee">
  ...
</cfcomponent>
```

Now that we have created a basic `Employee` component, we can call it from another CFML, `.cfm` template in a number of ways, firstly with the `<cfobject>` tag:

```
<cfobject component="Employee" name="myEmployee">
<cfdump eval=myEmployee>
```

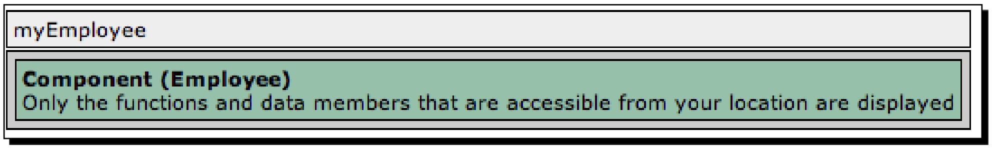
Another way is to use the `CreateObject()` function:

```
<cfset myEmployee = CreateObject("component", "Employee")>
<cfdump eval=myEmployee>
```

You can also use the "new" keyword to create your object:

```
<cfset myEmployee = new Employee()>
<cfdump eval=myEmployee>
```

All of the methods above will display the following output:

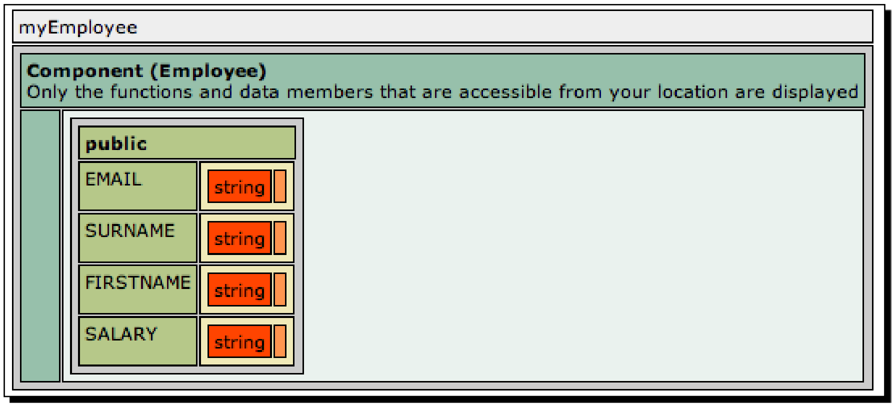


```
myEmployee
Component (Employee)
Only the functions and data members that are accessible from your location are displayed
```

So far, we don't see much in our Employee object, so let's change `Employee.cfc` to have a few properties:

```
<cfcomponent name="Employee">
  <cfset this.firstname = "">
  <cfset this.surname = "">
  <cfset this.salary = "">
  <cfset this.email = "">
</cfcomponent>
```

When we call this object now, we have properties that we can see, because we have used the `THIS` scope of the component, that means that the properties are public as can be seen if we output the result of instantiating the component.



```
myEmployee
Component (Employee)
Only the functions and data members that are accessible from your location are displayed
public
EMAIL      string
SURNAME    string
FIRSTNAME  string
SALARY     string
```

You can now set the properties directly, since they are public as follows:

```
<cfset myEmployee = new Employee()>
<cfset myEmployee.firstname = "John">
<cfset myEmployee.surname = "Smith">
<cfset myEmployee.salary = "20000">
<cfset myEmployee.email = "john.smith@somedomain.com">
<cfdump eval=myEmployee>
```

This is not the best way of accessing the properties of a component, so we can add functions to set and get the properties in a much safer, and even controlled manner. Let's add a getter and a setter for the `firstname` property:

```
<cfcomponent name="Employee">
  <cfset this.firstname = "">
  <cfset this.surname = "">
  <cfset this.salary = "">
  <cfset this.email = "">

  <cffunction name="getFirstName">
    <cfreturn this.firstname>
  </cffunction>

  <cffunction name="setFirstName">
    <cfargument name="firstname" type="string" required="true">
    <cfset this.firstname = arguments.firstname>
  </cffunction>
</cfcomponent>
```

You now see that we have used the `<cffunction>` tag to define two new functions. In the `getFirstName` function we simply use the `<cfreturn>` tag to return the `this.firstname` variable.

In the `setFirstName` function we use the `<cfargument>` tag to define what arguments the function can take, in this case, we are going to call it `firstname`, what type of argument it is (a string), and whether it is required. Inside a function, you can get all the arguments that are passed to that function in the `arguments` scope. In this case, we can refer to the string that is passed to our function as `arguments.firstname` and we can then set it to the `this` scope of the component.

We can now set the first name as follows:

```
<cfset myEmployee = new Employee()>
<cfset myEmployee.setFirstName("John")>
```

If we want to get the first name, we can now safely get it using the `getFirstName` function we defined:

```
<cfset myEmployee = new Employee()>
<cfset myEmployee.setFirstName("John")>
<cfset firstname =  myEmployee.getFirstName()>
```

The variable `firstname` in our template would now be set to "`John`".

Of course, in this component there are only a few properties and we would like to make sure they are set when we create the object; this is done by a special function called a constructor. Constructors are usually defined by a function with the name `init`, let's add that to our component:

```
<cfcomponent name="Employee">
  ...

  <cffunction name="init">
    <cfargument name="firstname" type="string" required="true">
    <cfargument name="surname" type="string" required="true">
    <cfargument name="salary" type="numeric" required="true">
    <cfargument name="email" type="string" required="true">

    <cfset this.firstname = arguments.firstname>
    <cfset this.surname = arguments.surname>
    <cfset this.salary = arguments.salary>
    <cfset this.email = arguments.email>
    <cfreturn this>
  </cffunction>

  ...
</cfcomponent>
```

As we can see in the previous code, we have created the `init` function and set all the parameters that we need to call when we create this component. The arguments passed into the `init` function are then set to the `this` scope, and we return the whole component back by using `<cfreturn this>` at the end of our function. We can now create our component in a few ways, by passing in the arguments in and using the `new` keyword:

```
<cfset myEmployee = new Employee("John", "Smith", "20000", "john.
smith@somedomain.com")>
<cfdump eval=myEmployee>
```

Or by calling the function after using the `CreateObject()` function:

```
<cfset myEmployee = CreateObject("component", "Employee")
   .init("John", "Smith", "20000", "john.smith@somedomain.com")>
<cfdump eval=myEmployee>
```

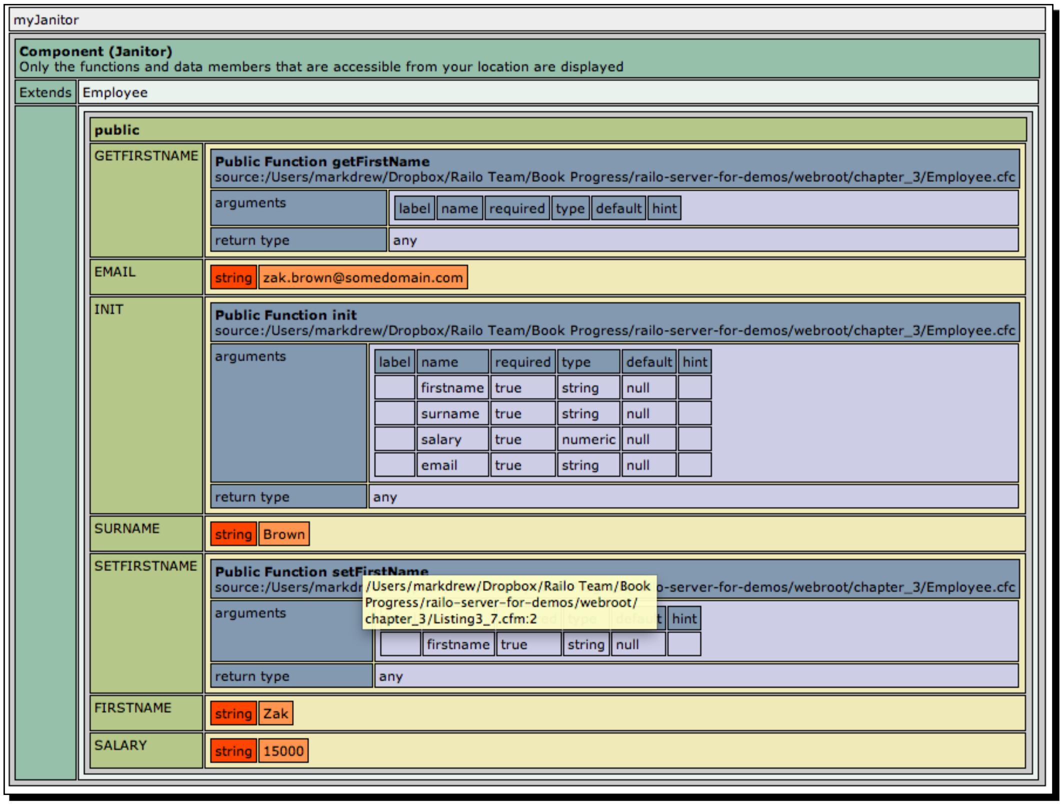
This gives us a component that is ready to do some work for us.

CFML Components also support inheritance. That is the ability of one object to "inherit" the properties and functions of another component. For example, what if we had a `Janitor` component? It might have all the properties and functions of an `Employee` object but it might also have very specific functions that an `Employee` component wouldn't have. Let's create a `Janitor.cfc` component that extends the `Employee` component:

```
<cfcomponent name="Janitor" extends="Employee">

</cfcomponent>
```

By using the `extends` attribute in the `<cfcomponent>` tag, we have now inherited from the `Employee` component, let's see what happens when we call our `Janitor` component:

```
<cfset myJanitor = new Janitor("Zak", "Brown", "15000", "zak.brown@
somedomain.com")>
<cfdump eval=myJanitor>
```



```
myJanitor
Component (Janitor)
Only the functions and data members that are accessible from your location are displayed
Extends  Employee
  public
  GETFIRSTNAME  Public Function getFirstName
                source:/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/chapter_3/Employee.cfc
                arguments     label | name | required | type | default | hint
                return type   any
  EMAIL         string  zak.brown@somedomain.com
  INIT          Public Function init
                source:/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/chapter_3/Employee.cfc
                arguments     label | name      | required | type    | default | hint
                                    | firstname | true     | string  | null    |
                                    | surname   | true     | string  | null    |
                                    | salary    | true     | numeric | null    |
                                    | email     | true     | string  | null    |
                return type   any
  SURNAME       string  Brown
  SETFIRSTNAME  Public Function setFirstName
                source:/Users/markdr /Users/markdrew/Dropbox/Railo Team/Book o-server-for-demos/webroot/chapter_3/Employee.cfc
                                     Progress/railo-server-for-demos/webroot/
                                     chapter_3/Listing3_7.cfm:2
                arguments     label | name      | required | type   | default | hint
                                    | firstname | true     | string | null    |
                return type   any
  FIRSTNAME     string  Zak
  SALARY        string  15000
```

You can see it has the same properties and methods as the Employee component.

We can now add functions that are specific to Janitors rather than all Employees, so let's add a "`clean`" function:

```
<cfcomponent name="Janitor" extends="Employee">
  <cffunction name="clean">
    <cfargument name="area" type="string" required="true">

    <cfreturn "I am cleaning the #arguments.area#">
  </cffunction>
</cfcomponent>
```

We can now ask the Janitor object to perform the `clean` function:

```
<cfset myJanitor = new Janitor("Zak", "Brown", "15000", "zak.brown@
somedomain.com")>
<cfoutput>#myJanitor.clean("hall")#</cfoutput>
```

This would display **I am cleaning the hall** in our browser.

What just happened?

Railo Components allow you to encapsulate data and functionality in a simple object that cleans up a lot of your code by making it reusable. You could have, of course, created a structure to do this, but it would have only stored the data and it would not have been able to run functions. This is the crux of object-oriented programming, creating encapsulated objects that represent some kind of real world object, both in their contents and behavior that you can use in your system.

Summary

This has been a tour de force chapter, we have learned:

- How tags and functions are used in Railo Server
- The various scopes that are available and where they can be seen and modified
- How to access databases with the `<cfquery>` tag as well as using `<cfstoredproc>` to call database stored procedures
- An introduction to Railo Server's object-oriented programming using components

There are many more things you can do with components in Railo Server, but hopefully this has given you some idea on how they behave. In the upcoming chapters, you will see how they can help you architect your application, control your application flow, and even allow you to create web services with ease.

4
Railo Server Administration

So far, we have looked at installing Railo Server, got to grips with the CFML language that we can write our templates in, and very briefly visited the Railo Administrator to create a datasource. In this chapter, we will explore the Railo Administrator in more detail and get to grips with:

- **Server and Web context**: What does each one manage and how do they affect each other
- **Settings**: How they affect performance, output, and internationalization of your server
- **Services**: How to access other services, such as databases, caches, event gateways, debugging, and mail servers
- **Extensions**: How you can add extra functionality to the Railo server through extensions
- **Archives and Resources**: How to access mappings, resources, custom tags, and CFX tags
- **Security**: Setting the password and access restrictions for each context

By the end of this chapter, we will have a good grasp of the various settings that can be changed and configured in the Railo Server Administration and how they affect the behavior of the server.

Let's get started!

Server and Web context

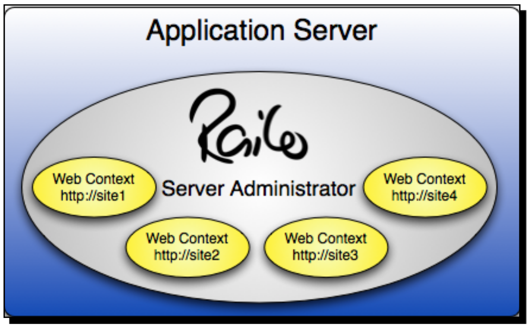
Before we delve too deeply into the administration of Railo Server, there is an important concept to get to grips with first. Even though, so far, we have been running only one website, Railo Server can actually run a number of websites.

A good example would be if we were running two different websites from our server, say our main website (`http://localhost:8888`) and another website, let's call it `http://site2.local:8888`.

Different websites might have different settings, and if you were an ISP, it probably would be maintained by different people. Railo Server allows you to do this by giving each context its own administrator and separating it from the other contexts, depending on which domain was used in the URL to access each site. These are called **Web contexts**.

Of course, the main administrator of the site should have access to all the settings and could also set "defaults" for the Web contexts. This is called the **Server context**.

You can visualize it as follows:



Imagine a server that has 20 websites installed in a single instance of Railo Server. We might want to add a new web context containing a database, some different custom tags, and a single search collection. In Railo Server, all these resources inside of a web context are private, and can only be used inside this single web context.

On the other hand, commonly requested Railo resources can be centrally defined in the Server Administrator, and therefore be made available to everyone.

If we want to give all the web contexts the ability to share the same mapping (let's say, `/coldbox` for the `ColdBox` framework), you can define it in the Railo Server Administrator. The mapping's definition is "read-only" for all web contexts. This definition can be seen in every local Web Administrator, but it cannot be deleted or modified by them. The only thing Railo allows you to do is to overwrite the global definition of the mapping by defining a local one with the same name.

This architecture implicates several advantages:

- **Web contexts are separated from one another**: The isolation of web contexts assures that no user of another web context goes "messing around" in another user's web context. Railo prevents this by default. System resources (libraries (JAR files), core) are shared amongst the server.

  The libraries used by Railo are shared between all web contexts. This helps reduce server memory and saves management and installation work.

- **Local administrator for each web context**: A server administrator can assign rights to specific users to manage their own context. This saves on change requests, and of course, gives power to the person that manages that context to change settings without affecting others.

- **Global definition of security rules**: A Railo Server Administrator can allow or disallow the installing and modification of certain resources that Railo Server provides. For example, the administrator could do the following:

  - Implement constraints on file access per web context
  - Set restrictions on Java-Access (including totally restricting access to base Java calls from CFML code)
  - Set restrictions on data source definitions
  - Create separate default settings for any web context

- **Restart and update**: An administrator can restart Railo Server without having to restart the whole servlet engine.

  An administrator is also able to easily update the server to the latest version, and even choose what patches/updates they want to install, be they stable, preview releases, or bleeding edge updates to test out.

- **Easy export of webs configuration files**: Railo Server stores all local settings of a web context in a single file inside the directory (usually in `/webroot/WEB-INF/railo/railo-web.xml.cfm`). This allows for the ability to zip up all the files in a web context, move them onto another server, and end up with the exact same settings as on the original server. Some of the settings might have become invalid because of invalid paths or different IP addresses. But at least they are in place.

Time for action – setting up an example context

Currently, our `http://localhost:8888` website points to the `<railo install>/webroot/` folder. Let's say we want to have the `http://site2.local:8888` website served from the `<railo install>/webroot2/` folder, and all the requests that go to the domain `http://site2.local:8888`, be served from there. We will need to create another context.

> Don't worry about the port number at the end of the domain (`8888`). We can change that in the configuration of Jetty, if you look in the folder you installed Railo Express; in the `/etc/` folder, you should see a file called `jetty.xml`. You can change the port by modifying the XML that says:
>
> <Set name="port"><Property name="jetty.port" default="8888"/></Set>
>
> to
>
> <Set name="port"><Property name="jetty.port" default="80"/></Set>
>
> And you will be running your websites without the need to add the port, since port `80` is the default port for websites.

To create a context, we are going to create the `http://site2.local:8888` domain on our local machine, create a folder where the webroot is going to be for `site2.local`, and add a context file to our Railo Server.

1. Let's add the domain to our hosts file. In OS X and Linux, you can do this by editing the `file /etc/hosts` (on Windows, this file is located in `C:\Windows\System32\drivers\etc\hosts`), and add the following to the bottom of the file:

    ```
    127.0.0.1     site2.local
    ```

2. This tells our computer that when we go to the `site2.local` domain, all our requests are pointed back to our machine.
3. Now create a folder in our `<railo install>` folder called `webroot2`.
4. Now when you have a look in the `<railo install>/contexts` directory, you will have a `railo.xml` file, which defines the default context.
5. Copy the `railo.xml` file to a new file called `site2.xml`.
6. Edit the `site2.xml` file and add the following code:

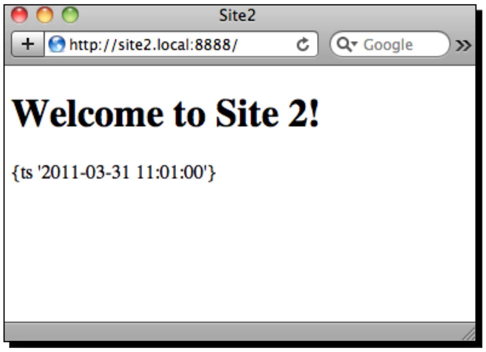
    ```
    <?xml version="1.0"  encoding="ISO-8859-1"?>
    <!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN"
      "http://www.eclipse.org/jetty/configure.dtd">
    <Configure class="org.eclipse.jetty.webapp.WebAppContext">
      <Set name="contextPath">/</Set>
      <Set name="resourceBase">
        <SystemProperty name="jetty.home" default="."/>/webroot2/
      </Set>
      <Set name="defaultsDescriptor">
        <SystemProperty name="jetty.home" default="."/>
        /etc/webdefault.xml
      </Set>
      <Set name="virtualHosts">
        <Array type="String"><Item>site2.local</Item></Array>
      </Set>
    </Configure>
    ```

7. The important lines comprise the `resourceBase` that has been changed to point to our `webroot2` folder and the entry in the `virtualHosts` section that points to our `site2.local`.

8. Create a file in `<railo install>/webroot2` and call it `index.cfm`. Add the following in the file, so we can see that everything is working:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Site2</title>
  </head>
  <body>
    <h1>Welcome to Site 2!</h1>
    <p><cfoutput>#Now()#</cfoutput></p>
  </body>
</html>
```

9. Once you save the file, head over to `http://site2.local:8888/`, and you should now see your **Site2** working.



Welcome to Site 2!

{ts '2011-03-31 11:01:00'}

What just happened?

By adding an entry in the hosts file, we have been able to create a domain that our local machine will respond to. Once we did that, we were able to create a new context that is separate from our main context, which will serve pages from a different directory, and only respond to pages on the `http://site2.local:8888` domain. Now, we can go ahead and check out the differences between a Railo Server Administrator and a Railo Web Administrator.

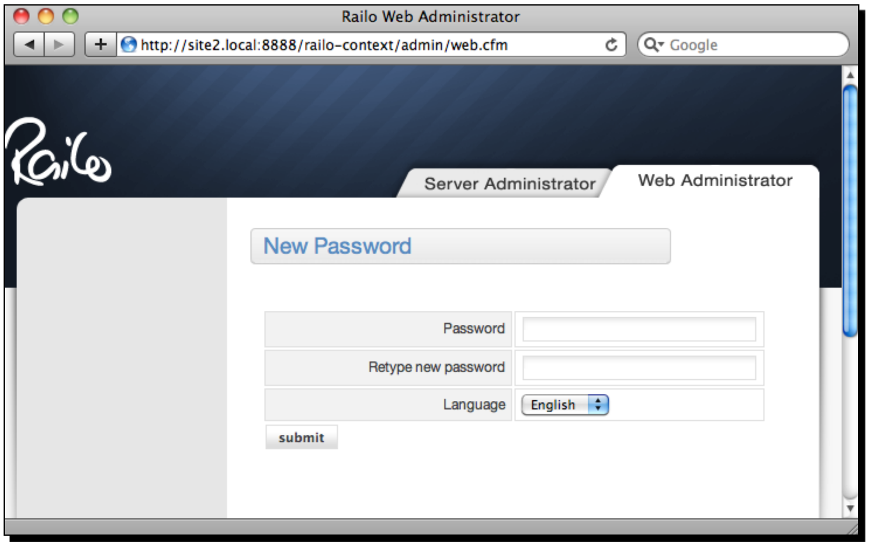
Setting up security

Now that we have set up the second context, we can access the administration through the URL `http://site2.local:8888/railo-context/admin/web.cfm`. Let's have a proper look this time.

Time for action – setting your password

Let's go and set our passwords for the Railo Web administrator (since we already set our Railo Server Administrator password in Chapter 2, *Installing Railo Server*) of our new context:

1. In the browser, go to `http://site2.local:8888/railo-context/admin/web.cfm`.
2. You will see the **New Password** screen with the **Web Administrator** tab highlighted.



3. Enter a **Password** and then retype it in the dialog. You should now see the **Railo Web Administrator** homepage.
4. Now that you have logged in, click on the **Logout** button to get back to the **Web Administrator** login page.
5. You will now see the **Login** prompt, which you can now use to log in using the password you set previously.

What just happened?

Railo Server allows you to set a different password for the Web Administrator in each context. We have just set the password for the `site2.local` (`http://site2.local:8888/railo-context/admin/web.cfm`) context, which is separate from the context at `http://localhost:8888/railo-context/admin/web.cfm`.

You can follow the same procedure to change the main Server Administrator by clicking on the **Server Administrator** tab.

How contexts relate to each other

Now that we have set up the passwords for our Web and Server Administrator sections, we can investigate how changes to settings in one context affects the settings in another.

Let's take an example of having two contexts. One hosts a website that is aimed at users from Hong Kong and the other is aimed at users from the USA. It would make no sense if the time zone settings (such as the time) would be the same for both. Normally, this setting would be defined by the actual geographic location of the whole server. Railo Server allows you to customize this.

Let's check it out.

Time for action – setting the time zone

Let's create a file that will display the time zone for the current context:

1. In your `<Railo Intall Directory>/webroot/`, create a file called `server_timezone.cfm`.

2. In that file, enter the following code:

   ```
   <cfoutput>#getTimeZone()# #Now()#</cfoutput>
   ```

3. Let's copy this file to our second context located in `<Railo Intall Directory>/webroot2/`.

4. If you now go to `http://localhost:8888/server_timezone.cfm`, you should see your local time zone, for example, mine displays:

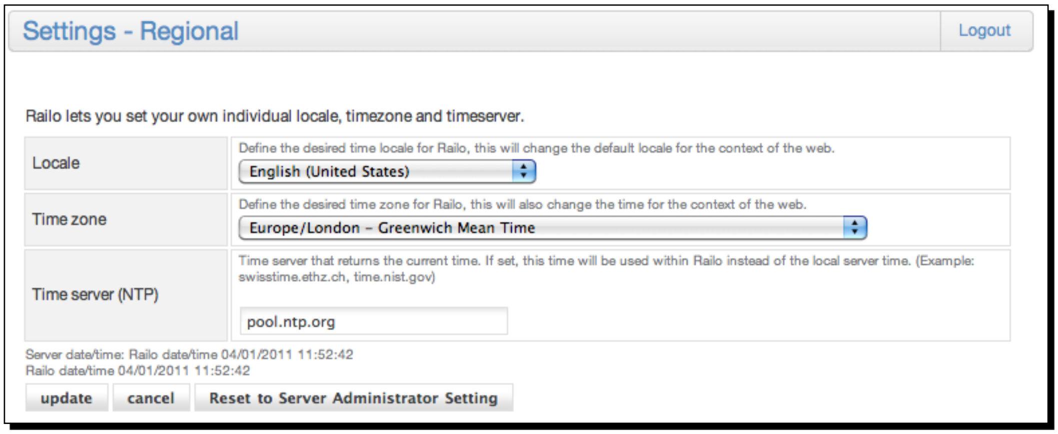
   America/New_York {ts '2011-09-16 13:08:10'}

5. If you now go to your second context by going to `http://site2.local:8888/server_timezone.cfm`, you should also see:

   America/New_York {ts '2011-09-16 13:10:27'} (or at least the same as was displayed in your first context).

6. Let's set the server context and see what happens. In your browser, go to `http://site2.local:8888/railo-context/admin/server.cfm` and log in (or enter a new password and confirm it if it hasn't already been set).

7. On the left menu, under **Settings**, click on the **Regional** link, and you will see the **Settings - Regional** screen, as shown in the following screenshot:



Settings - Regional

Railo lets you set your own individual locale, timezone and timeserver.

Locale	Define the desired time locale for Railo, this will change the default locale for the context of the web. English (United States)
Time zone	Define the desired time zone for Railo, this will also change the time for the context of the web. Europe/London - Greenwich Mean Time
Time server (NTP)	Time server that returns the current time. If set, this time will be used within Railo instead of the local server time. (Example: swisstime.ethz.ch, time.nist.gov) pool.ntp.org

Server date/time: Railo date/time 04/01/2011 11:52:42
Railo date/time 04/01/2011 11:52:42

8. In the **Time zone** drop-down box, change the setting to (for example) **America/Jamaica**. This is now the default setting for the whole server, and therefore all the contexts. Click **Update** to confirm these settings.

9. If you now reload the pages at `http://localhost:8888/server_timezone.cfm` and `http://site2.local:8888/server_timezone.cfm`, you will see that the server-wide settings have been applied to all the contexts, as the pages now display **America/Jamaica {ts '2011-09-16 12:11:29'}**.

10. Let's change the first context to the **UK** first. Let's head to the Web Administrator for our **localhost** context by going to `http://localhost:8888/railo-context/admin/web.cfm`, and then clicking on the **Settings – Regional** (as you previously did for the server).

11. In the **Settings – Regional** page for the **localhost** web context, let's select the **Europe/London – Greenwich Mean Time** time zone and click on **Update**.

12. When we now reload `http://localhost:8888/server_timezone.cfm`, we can see that the time zone displayed is now **Europe/London**, and if we also reload **http://site2.local:8888/server_timezone.cfm**, we can see that the server-wide settings still apply to this context, as it is still displaying **America/Jamaica**.

13. Let's change the `site2.local` time zone now by following the same procedure as **localhost**, but by going to `http://site2.local:8888/railo-context/admin/web.cfm`, selecting **Settings – Regional**, changing the time zone to **Asia/Hong Kong – Hong Kong Time** in the drop-down, and clicking the **Update** button.

14. Now when we reload each of the time zone files in each context, we will get **Europe/London** for the **localhost** context and **Asia/Hong_Kong** for the `site2.local` context.

What just happened?

The settings that are made in the Railo Server Administrator apply to all existing and new contexts. These can be seen as the default settings. If you want to override these settings, you can make changes in the Railo Web Administrator, which will only affect the current context. This is very handy for settings that are specific to an application.

The Railo Web Administrator

In the Railo Web Administrator, you can change many settings regarding how you want the code in the context to behave, what resources it has access to, how output should be handled, and so on.

The Railo Web Administrator is organized into several sections, which are listed as follows:

- Settings
- Services
- Extension
- Remote
- Archives and Resources
- Development
- Security
- Documentation

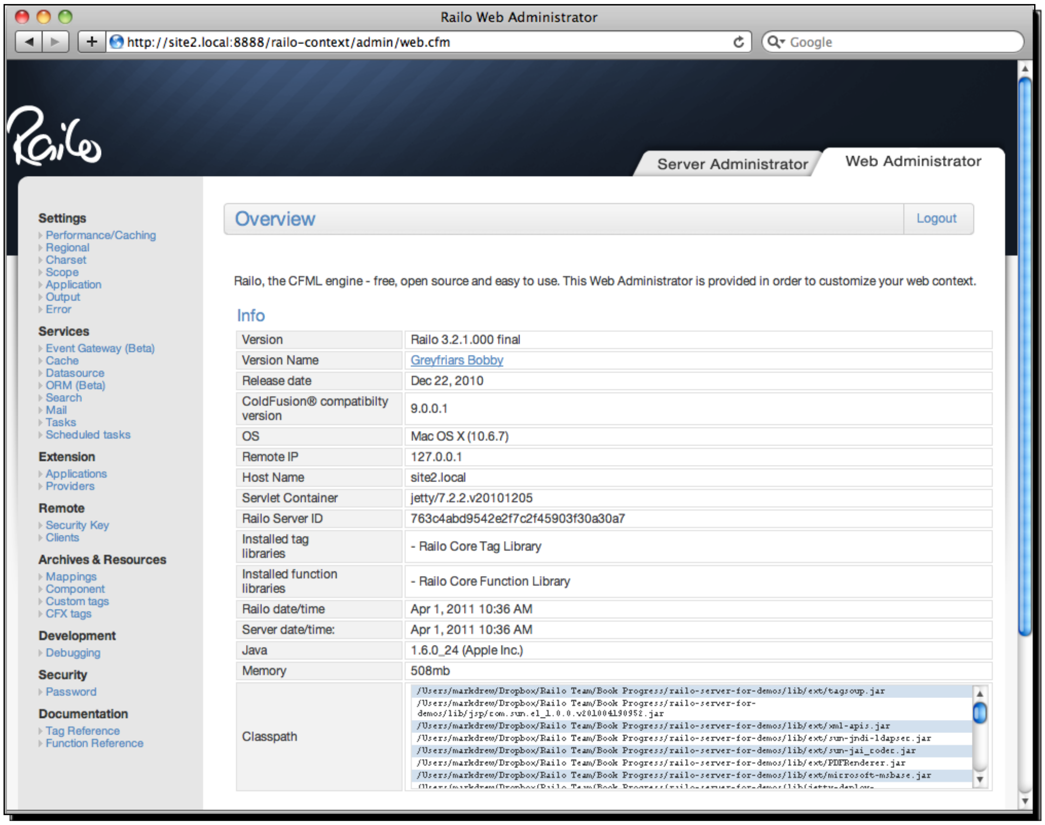
These menu items are likely to change, depending on which plugins and extensions you have installed (we shall go into Railo Extensions in detail later in the book), but the main structure is set.

Remember, any changes that you make to the Railo Web Administrator will only affect the current context. If you want to make changes across all the contexts, you can do them in the Server Administrator.

Time for action – investigating the Web Administrator

Now that we have secured the Web Administrator, let's log in again and explore it:

1. In your browser, go to `http://site2.local:8888/railo-context/admin/web.cfm`.
2. Enter your password and click on the **submit** button.
3. You should now see the **Railo Web Administrator** - **Overview**, as shown in the following screenshot:



Railo, the CFML engine - free, open source and easy to use. This Web Administrator is provided in order to customize your web context.

Info

Version	Railo 3.2.1.000 final
Version Name	Greyfriars Bobby
Release date	Dec 22, 2010
ColdFusion® compatibilty version	9.0.0.1
OS	Mac OS X (10.6.7)
Remote IP	127.0.0.1
Host Name	site2.local
Servlet Container	jetty/7.2.2.v20101205
Railo Server ID	763c4abd9542e2f7c2f45903f30a30a7
Installed tag libraries	- Railo Core Tag Library
Installed function libraries	- Railo Core Function Library
Railo date/time	Apr 1, 2011 10:36 AM
Server date/time:	Apr 1, 2011 10:36 AM
Java	1.6.0_24 (Apple Inc.)
Memory	508mb
Classpath	/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/lib/ext/tagsoup.jar /Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/lib/jsp/com.sun.el_1.0.0.v201004190952.jar /Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/lib/ext/xml-apis.jar /Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/lib/ext/sun-jndi-ldapsec.jar /Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/lib/ext/sun-jai_codec.jar /Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/lib/ext/PDFRenderer.jar /Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/lib/ext/microsoft-msbase.jar /Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/lib/jetty-deploy-

What just happened?

Using the password we set earlier, we have logged into the Railo Web Administrator and seen the **Overview** page.

This **Overview** page gives us a lot of important information about our current installation such as:

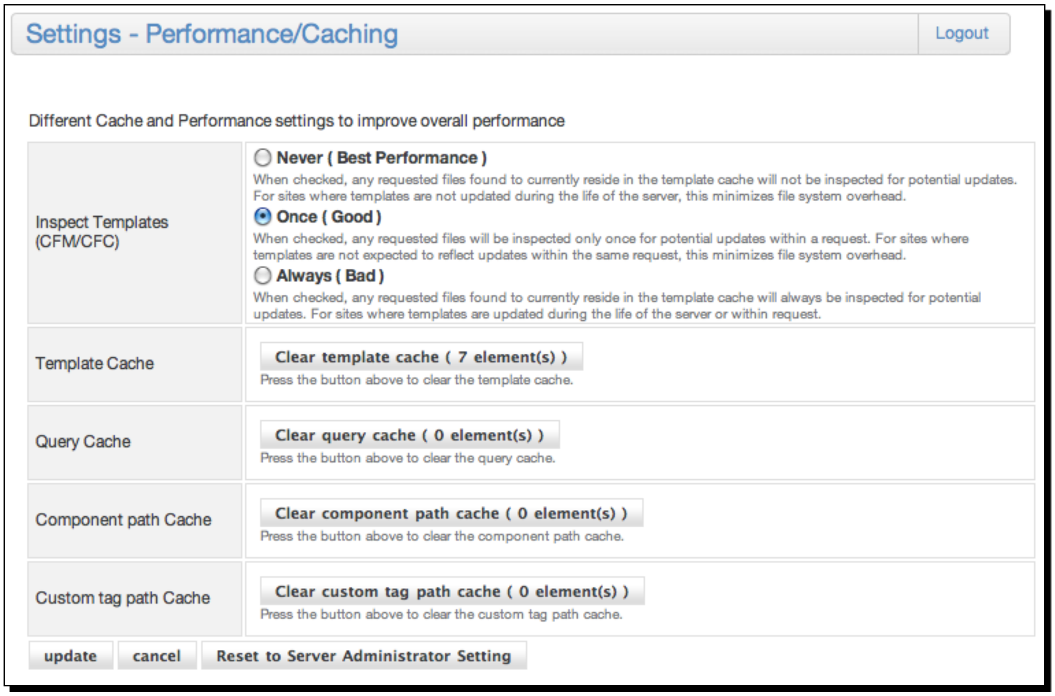
- Railo version, name, and release date
- The Adobe ColdFusion(tm) compatibility level
- The operating system
- IP address of the caller and the host name
- The Servlet container type and server ID
- Loaded tag libraries
- Loaded function libraries
- Local Web Administrator time
- Server time
- Loaded **Java Virtual Machine** (**JVM**) version
- Maximum available amount of memory
- Class path (loaded Java archives JARs)
- Railo Company information

Settings

The settings are general settings that are controlling the behavior of Railo Server. These affect things such as template caching, regional settings, how components are handled, character sets, and other issues. Let's look at each section in detail.

Performance/Caching

The settings you make in the **Performance/Caching** section determine how Railo Server deals with changed CFM (and CFC) files. Normally, Railo will compile any modified file into a new Java class file. While this is very fast (depending on the complexity and size of the file, this might take around 10 ms), each `<cfinclude>`, `<cfmodule>`, CFC invocation, Custom Tags call, and so on, will check whether the related file has changed. This file checking will only take a few milliseconds at the OS level, but this might add up to give you some quite heavy performance problems.



Settings - Performance/Caching

Different Cache and Performance settings to improve overall performance

Inspect Templates (CFM/CFC)	○ **Never (Best Performance)** When checked, any requested files found to currently reside in the template cache will not be inspected for potential updates. For sites where templates are not updated during the life of the server, this minimizes file system overhead. ● **Once (Good)** When checked, any requested files will be inspected only once for potential updates within a request. For sites where templates are not expected to reflect updates within the same request, this minimizes file system overhead. ○ **Always (Bad)** When checked, any requested files found to currently reside in the template cache will always be inspected for potential updates. For sites where templates are updated during the life of the server or within request.
Template Cache	Clear template cache (7 element(s)) Press the button above to clear the template cache.
Query Cache	Clear query cache (0 element(s)) Press the button above to clear the query cache.
Component path Cache	Clear component path cache (0 element(s)) Press the button above to clear the component path cache.
Custom tag path Cache	Clear custom tag path cache (0 element(s)) Press the button above to clear the custom tag path cache.

For the **Inspect Templates (CFM/CFC)** section, the following settings are available:

- **Never (Best Performance)**: If selected, each template requested, which is in the template cache, will not be checked for potential updates. For sites where the templates do not change during the server runtime, this setting minimizes the system load.

  Select this option on a live server, where you know that the templates used almost never change. If a template is updated, you can simply flush the template cache by clicking on the **Clear template cache** button, which will force the server to recompile the files. You can also use the function `pagePoolClear()` to flush the template cache.

- **Once (Good)**: If selected, the templates are only checked once per request for potential updates. For sites where the templates do not change very often during the server runtime, this setting reduces the system load.

  We would suggest that this is the most recommended setting, because it is a very good mix between keeping templates up-to-date and performing well. This is also the default setting.

- **Always (Bad)**: If selected, each template that is in the template cache will always be checked for potential updates. This is the best choice for sites where the templates might change during a single request. If you cannot choose the **Never** or **Once** settings, because files might be generated or recreated by an application during the execution of a request, this option may be a possibility. But, if possible, this should be avoided on production systems. In order to clear the page pool if files are generated, you can always call the `pagePoolClear()` function.

Time for action – comparing template caching settings

Let's have a look at what impact these settings have on a normal request.

This way we can compare how long it takes for the request for each of the different settings. Let's create some templates to test:

1. In the `<Railo Install Directory>/webroot/`, create a template called `speed_test.cfm` with the following code:

```
<cfscript>
  totalPage = 0;
  loop from=1 to=10 index="i" {
    FileWrite("my_include.cfm", "<cfset myvar_#i# = Now()>");
    includeStart = getTickCount();
    include template="my_include.cfm";
    includeTotal = getTickCount() - includeStart;
    totalPage += includeTotal;
    WriteOutput("Include #i# took: #includeTotal# <br>");
    sleep(1000);
  }
  WriteOutput("Total include time #totalPage#");
</cfscript>
```

2. This code does a simple loop from `1` to `10`, and each time, it writes the code `<cfset myvar_#i# = Now()>` into a file called `my_include.cfm`. Essentially, this will be rendered as `<cfset myvar_1 = Now()>` for the first loop into the template, and basically change the actual code for each loop.

3. Then, we time how long it takes to include that file in our main template. The other parts of the code simply time how long each `include` takes using the `getTickCount()` function.

4. Before we run this, we should go to the Railo Web Administrator by going to `http://localhost:8888/railo-context/admin/web.cfm`.

5. Click on the **Performance/Caching** link on the left, then select **Always (Bad)**, and press the **update** button.

6. When we run the template by going to `http://localhost:8888/speed_test.cfm` a few times, we will get something like (your results may vary of course):

   ```
   Include 1 took: 2
   Include 2 took: 6
   Include 3 took: 12
   Include 4 took: 12
   Include 5 took: 2
   Include 6 took: 2
   Include 7 took: 1
   Include 8 took: 1
   Include 9 took: 1
   Include 10 took: 2
   Total include time 41
   ```

7. Then, if we go back to the Railo Web Administrator, select **Once (Good)**, click the **update** button, then re-run the template, and we will get some results similar to the ones that follow:

   ```
   Include 1 took: 27
   Include 2 took: 0
   Include 3 took: 0
   Include 4 took: 0
   Include 5 took: 0
   Include 6 took: 0
   Include 7 took: 0
   Include 8 took: 0
   Include 9 took: 0
   Include 10 took: 0
   Total include time 27
   ```

8. Finally, if you now select **Never (Best Performance)** and reload our speed test a couple of times, you should get the following results:

```
Include 1 took: 0
Include 2 took: 0
Include 3 took: 0
Include 4 took: 0
Include 5 took: 0
Include 6 took: 0
Include 7 took: 0
Include 8 took: 0
Include 9 took: 0
Include 10 took: 0
Total include time 0
```

What just happened?

In the previous examples, we can see how the **Inspect Templates** setting in the **Performance/Caching** section of the Railo Web Administrator is changing and how Railo Server deals with templates on each request.

In the first example, each time we tried to include `my_template.cfm`, Railo Server re-parsed and compiled the template.

When we set the performance to **Once (Good)**, the first time the template was included, it was inspected for changes. On subsequent requests, it took no time, since Railo Server already had compiled it and could now ignore it.

Finally, when we have the setting of **Never (Best Performance)**, Railo Server checks if it has a compiled version of that template; if it has, it never checks for changes again. Hence we get zero milliseconds for each `include`.

This might be a good thing on some systems, but remember that the template has indeed changed, so in some cases, we do want Railo Server to actually re-check for changes. In this case, you can run the `pagePoolClear()` function, which will remove the templates from the **Template Cache** and then reload them again.

In addition to determining the way Railo Server treats CFML templates that are in the template cache, you can additionally flush the template and query cache. The Web Administrator will always display the number of items that are currently stored in the cache. If you click on the **clear cache** button, the items are removed. Don't be surprised that some elements reappear in the template cache. In order to flush the cache, some templates that land in this cache had to be called.

The Server Administrator determines what the standard caching method is, which of course, you can overwrite in each of the Web Administrators. This is why you have a button **Reset to Server Administrator Setting** that will reset any changes you have made to the value set in the Server Administrator.

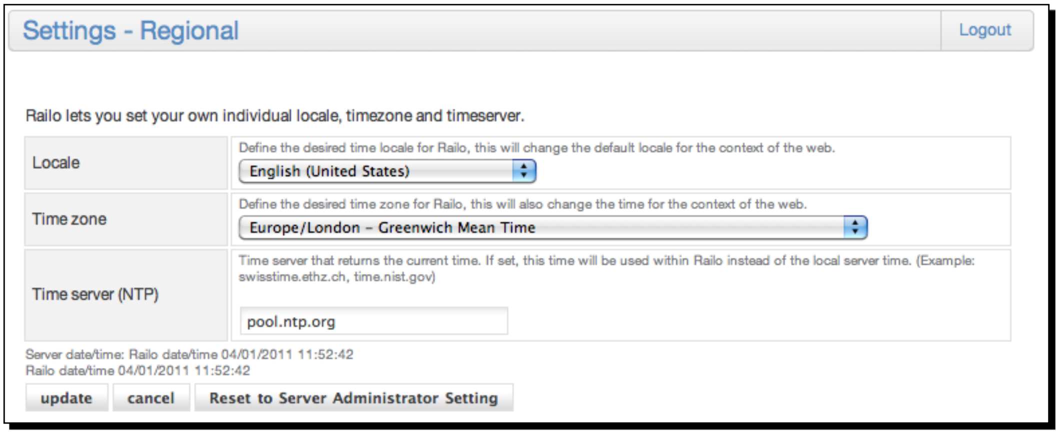
Regional

We have already seen this section in our previous example, dealing with the differences between contexts and the Server context, and have used the Locale and time zone features.

Besides setting the current locale for display, it also allows us to access locale-specific formatting of values, such as dates and numbers. This is the default region that will be used, if we use a conversion function `LSNumberFormat()` or `LSDateFormat()`. Once set, you never have to adjust this value.

Of course, there are times that you want to get the time defined by the server itself. Railo Server has a **Built in Function** (**BIF**) called `nowServer()`. This function returns the current server time. You can easily test this. Just change the time zone and click on **update**. After that, you will see that the regional time has changed accordingly (Server time/Railo time).

Railo Server synchronizes its time with a time server. You can enter a valid URL of the desired time server in the **Time Server (NTP)** field. Railo Server will then update its time every hour from the time server defined. In order to use a time server, the server only needs to support the **Network Time Protocol** (**NTP**).



Railo lets you set your own individual locale, timezone and timeserver.

Locale	Define the desired time locale for Railo, this will change the default locale for the context of the web. English (United States)
Time zone	Define the desired time zone for Railo, this will also change the time for the context of the web. Europe/London – Greenwich Mean Time
Time server (NTP)	Time server that returns the current time. If set, this time will be used within Railo instead of the local server time. (Example: swisstime.ethz.ch, time.nist.gov) pool.ntp.org

Server date/time: Railo date/time 04/01/2011 11:52:42
Railo date/time 04/01/2011 11:52:42

Have a go hero – get the server time zone

Why don't you try it yourself? Modify the code we used previously when choosing the time zones for each context and display the server time.

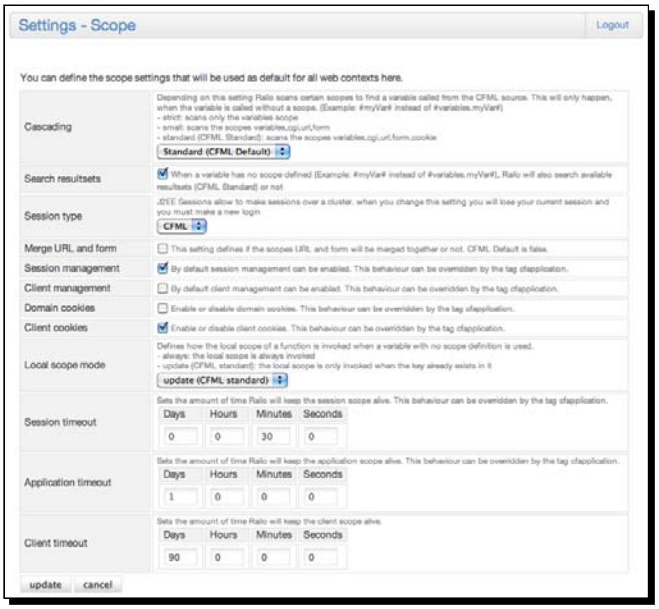
Charset

The *Charset* section allows you to set the different character sets that are used for different purposes. The default is the one set by the operating system and the Java charset.

Charset type	Description
`Template charset`	Defines the charset used when reading CFML templates from the file system. It is the charset that is pre-defined by the Operating System. (For Windows, it normally is CP1252).
`Web charset`	Defines the charset used in forms and for the output in the browser. The charset is predefined by the charset of the used JRE.
`Resource charset`	Defines the character set for reading from/writing to various resources, such as ZIP files.

Scope

Scopes are special structures that store information within the Railo Server. This section allows you to modify different behaviors of the scopes, such as how lookups of variables are handled, the type of sessions used by the context, enabling session, cookie, and client scope management, and the time-out values for various scopes.



You can define the scope settings that will be used as default for all web contexts here.

Cascading: Depending on this setting Railo scans certain scopes to find a variable called from the CFML source. This will only happen, when the variable is called without a scope. (Example: #myVar# instead of #variables.myVar#)
- strict: scans only the variables scope
- small: scans the scopes variables,cgi,url,form
- standard (CFML Standard): scans the scopes variables,cgi,url,form,cookie

Standard (CFML Default)

Search resultsets: ☑ When a variable has no scope defined (Example: #myVar# instead of #variables.myVar#), Railo will also search available resultsets (CFML Standard) or not

Session type: J2EE Sessions allow to make sessions over a cluster. when you change this setting you will lose your current session and you must make a new login

CFML

Merge URL and form: ☐ This setting defines if the scopes URL and form will be merged together or not. CFML Default is false.

Session management: ☑ By default session management can be enabled. This behaviour can be overridden by the tag cfapplication.

Client management: ☐ By default client management can be enabled. This behaviour can be overridden by the tag cfapplication.

Domain cookies: ☐ Enable or disable domain cookies. This behaviour can be overridden by the tag cfapplication.

Client cookies: ☑ Enable or disable client cookies. This behaviour can be overridden by the tag cfapplication.

Local scope mode: Defines how the local scope of a function is invoked when a variable with no scope definition is used.
- always: the local scope is always invoked
- update (CFML standard): the local scope is only invoked when the key already exists in it

update (CFML standard)

Session timeout: Sets the amount of time Railo will keep the session scope alive. This behaviour can be overridden by the tag cfapplication.

Days	Hours	Minutes	Seconds
0	0	30	0

Application timeout: Sets the amount of time Railo will keep the application scope alive. This behaviour can be overridden by the tag cfapplication.

Days	Hours	Minutes	Seconds
1	0	0	0

Client timeout: Sets the amount of time Railo will keep the client scope alive.

Days	Hours	Minutes	Seconds
90	0	0	0

Cascading

One of the benefits of CFML is also one of the determents, something called scope cascading. CFML makes it easy for a programmer to use variables without identifying which scope they are stored in.

Let's think of an analogy of calling the register in school. Let us assume that our application is a school and all pupils are variables. If you call out the name *Peter* in a classroom, without telling which Peter you mean, you might get lucky if there is only one Peter in the room and only receive one response. But if there is no Peter in the class, then you have to look for a Peter. You use a certain order in which you scan the classrooms. You start with the first classrooms on your left and continue until you have found a Peter. So you call *Peter* and CFML gets you a Peter. Sometimes surprising results happen. Maybe the janitor named Peter turns up, although you were in fact looking for *Peter Parker* from *class 7-b*. So why weren't we required to call *Peter Parker*, *class 7-b come here!!!*? CFML saves you the trouble. You don't have to do it specifically. CFML leaves you the freedom to not strictly scope your variables. It does come with the price that you might end up with a different Peter than you expected.

Getting back to programming, you can configure how Railo Server handles these look-ups to the point that "scope cascading" ("Get me a Peter") can be even turned off. If it is turned off, you need to write more specific code to reference your variables.

Time for action – restricting the scoping of variables

Let's have a look at examples of these settings in practice.

1. In your `<Railo Install Directory>/webroot`, create a template called `scope_test.cfm`. In this template, just add the following code:

   ```
   <cfoutput>#name#</cfoutput>
   ```

2. Let's load this template by calling `http://localhost:8888/scope_test.cfm?name=Mark`. Notice that we have added a URL variable called `name` with the value of `Mark`.
3. The output we get for this template is still `Mark`, since Railo looked through our scopes, starting with the `VARIABLES` scope, then the `CGI`, `URL`, `FORM`, and finally `COOKIE`, and found the variable `name` in the `URL` scope.
4. If we go to the Railo Web Administrator, then go to the **Scope** section, and change the cascading from **Standard (CFML Default)** to **Small**, the code will still work, since Railo Server has only had to look through the `VARIABLES`, `CGI`, `URL`, and `FORM` scopes.

5. Let's change the cascade to **Strict** and reload the template in the browser. This time, we get an error that says:

   variable [NAME] doesn't exist

6. Let's change our code to be a bit more precise; change the code in the `scope_test.cfm` template to have a scope:

   ```
   <cfoutput>#URL.name#</cfoutput>
   ```

7. When we reload the template, we see that it now works.

What just happened?

Railo Server lets you access variables without having to prefix the scope they live in quite nicely. Of course, this has performance issues, since Railo Server will have to look through various scopes, until it finds the variable. If there is a variable with the same name, then it will return the first one it finds.

You find that you have better code if you are more explicit with the name of the scope and then the variable, as in the `<scope>.<variable>` format, for example `URL.name`.

Using stricter scoping of variables will invariably speed up your application, as Railo Server has to do less work in looking up variables. It will also eliminate potential issues when the same variable name is used in multiple scopes.

Session type

Railo Server is fast, but sometimes heavy traffic slows down even on the fastest system. It could become necessary to cluster multiple servers running Railo Server together. Railo Server offers the possibility to cluster two or more servers in a JEE environment. In this case, the JEE server handles the sessions, and you can define the session type in the Railo administrator. Possible values are `JEE` or `CFML`. This would allow the underlying Servlet Container (such as Resin, Tomcat, or Jetty) to manage the `SESSION` scope across the cluster.

Session identifiers that are passed through the URL have a higher priority when using CFML Session. It is the other way round when you use JEE sessions. In such cases, cookies are preferred.

Combine the URL and Form scope

If we enable the **Merge URL and FORM scopes** setting, Railo Server merges the variables from the `FORM` and `URL` scope into a single scope. The variables can then be accessed either over the `FORM` or the `URL` scope.

Variables with the same name that are present in both scopes are combined into a comma separated list, as if a variable has been passed twice in the form scope.

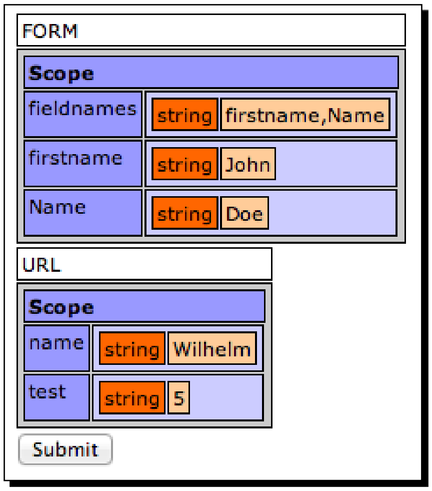
Time for action – merging the URL and FORM scopes

1. Let's test this out. Create a form called `scope_merge.cfm` and add the following code:

```
<cfif isDefined("form.firstname")>
  <cfdump eval=FORM>
  <cfdump eval=URL>
</cfif>
<form action="scope_merge.cfm?name=Wilhelm&test=5" method="post">
  <input type="Hidden" name="firstname" value="John">
  <input type="Hidden" name="Name" value="Doe">
  <input type="Submit">
</form>
```

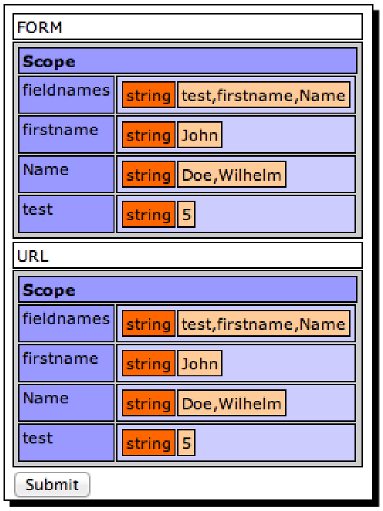
2. This code creates a form that will submit and create `FORM` variables that are hidden, as well as `URL` variables.

3. When we call this template, by going to `http://localhost:8888/scope_merge.cfm`, we get a simple form which, when submitted, gives the following output:



```
FORM
Scope
fieldnames   string  firstname,Name
firstname    string  John
Name         string  Doe
URL
Scope
name   string  Wilhelm
test   string  5
```

4. Now, if we go to the Railo Web Administrator and tick the checkbox next to the **Merge URL and FORM** section, we would get something similar to the following screenshot:



FORM

Scope		
fieldnames	string	test,firstname,Name
firstname	string	John
Name	string	Doe,Wilhelm
test	string	5

URL

Scope		
fieldnames	string	test,firstname,Name
firstname	string	John
Name	string	Doe,Wilhelm
test	string	5

What just happened?

Railo Server allows you to join up the URL and FORM scopes and copy the values into both scopes. If you have values that are named the same, they will be joined in a comma-delimited list, as if they were the same named field in a form.

Session management

Railo Server allows sessions to be used. Railo Server keeps a session for each visitor of an application. In order to recognize a returning user, you need a few tricks. You could either use URL parameters, hidden form fields, or cookies, in order for Railo Server to address an already existing session to the returning user. When using CFML sessions, Railo Server creates two IDs (named `CFID` and `CFTOKEN`), which the user sends with the request. These IDs are used to assign the right session to the user. In order to use sessions, you need to turn on **session management** in the Railo Web Administrator. Sessions are quite handy when you like to store some data during the visit of a user (shopping cart, user data, preferences, and so on).

The workflow when initializing a session with cookies goes as follows:

- A user calls a website (for example, `http://www.getrailo.com`).
- Since Railo Server is running on this system and **session management** is turned on for the requested application, the browser stores (if possible) a cookie at the end of the requests on the client's local PC.
- With every subsequent request from this client, the cookies will be sent back to the server.
- Railo Server then identifies the user and assigns the corresponding session to the user.

- Railo Server creates a session only when it is used.
- The session is stored in the server's memory. So, for example, if we set a value of 5 in a user's session with the following code:

  ```
  <cfset session.retry = 5>
  ```

  The value of 5 will be stored in the variable session.retry. The next time the user calls a page (as long as it's in the same application), the value of the session.retry variable will still be available in the session scope.

Client management

This option allows storing data of an application user in the client scope. The default usage of the client scope can be turned on here. Client variables are like an extended cookie.

Domain cookies

When you enable this option, the cookies stored by Railo Server are stored for a complete domain and not only for a single host.

Client cookies

By turning on this option, Railo Server automatically stores cookies on the client computer, when using sessions.

Session timeout

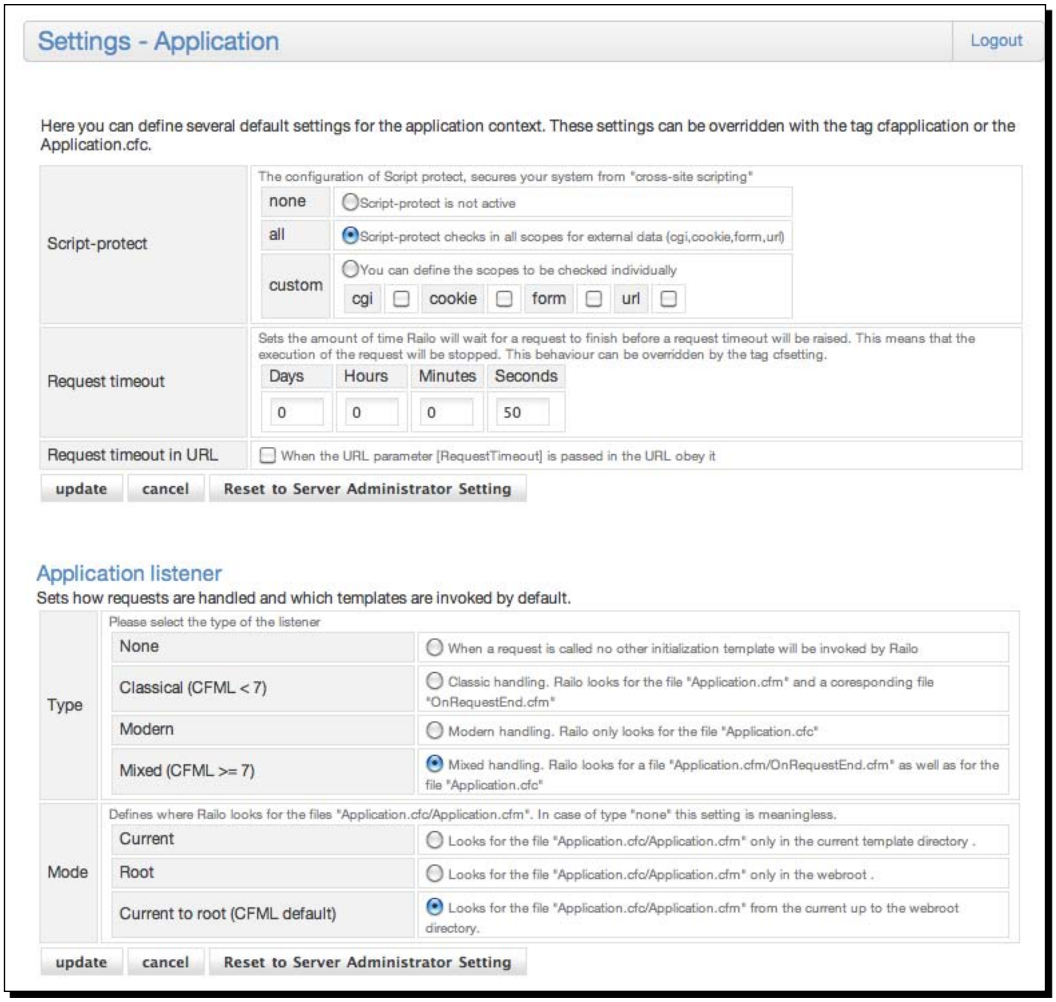
This option defines the time span the session scope is kept alive for a certain user. If during this time span there is no activity in this user's session, the session variable will be deleted from the server's memory. Higher time span values consume more memory on the server.

Application timeout

This option defines the time span the application scope is kept alive. If during this time span there is no activity in this application, the application scope will be deleted from the server's memory. Higher time span values consume more memory on the server.

Application

Under **Application**, you can define several different application-specific settings. We shall go into more detail about the application lifecycle and how it is defined in Railo Server in the next chapter, but for the moment, you should be aware that this is where you can change how Railo Server deals with different aspects of applications.



Settings - Application

Here you can define several default settings for the application context. These settings can be overridden with the tag cfapplication or the Application.cfc.

Script-protect	The configuration of Script protect, secures your system from "cross-site scripting" none ○ Script-protect is not active all ● Script-protect checks in all scopes for external data (cgi,cookie,form,url) custom ○ You can define the scopes to be checked individually cgi □ cookie □ form □ url □
Request timeout	Sets the amount of time Railo will wait for a request to finish before a request timeout will be raised. This means that the execution of the request will be stopped. This behaviour can be overridden by the tag cfsetting. Days 0 Hours 0 Minutes 0 Seconds 50
Request timeout in URL	□ When the URL parameter [RequestTimeout] is passed in the URL obey it

Application listener

Sets how requests are handled and which templates are invoked by default.

Type	Please select the type of the listener	
	None	○ When a request is called no other initialization template will be invoked by Railo
	Classical (CFML < 7)	○ Classic handling. Railo looks for the file "Application.cfm" and a coresponding file "OnRequestEnd.cfm"
	Modern	○ Modern handling. Railo only looks for the file "Application.cfc"
	Mixed (CFML >= 7)	● Mixed handling. Railo looks for a file "Application.cfm/OnRequestEnd.cfm" as well as for the file "Application.cfc"
Mode	Defines where Railo looks for the files "Application.cfc/Application.cfm". In case of type "none" this setting is meaningless.	
	Current	○ Looks for the file "Application.cfc/Application.cfm" only in the current template directory .
	Root	○ Looks for the file "Application.cfc/Application.cfm" only in the webroot .
	Current to root (CFML default)	● Looks for the file "Application.cfc/Application.cfm" from the current up to the webroot directory.

Script Protect

This setting should protect your application from "cross-site scripting" attacks. Here, you can choose to switch it off (not recommended). The setting protects all scopes that are checked for the inclusion of the `<script>` tag or specific scopes.

Cross-site scripting

Cross-site scripting (**XSS**) is a type of computer security vulnerability, typically found in web applications, which allows code injection by malicious web users into the web pages viewed by other users. Examples of such code include HTML code and client-side scripts. An exploited cross-site scripting vulnerability can be used by attackers to bypass access controls, such as the same origin policy.

If you turn on **script protection**, Railo Server checks incoming scopes for foreign data and replaces it with harmless data. When you have set the setting to all the scopes, `CGI`, `COOKIE`, `FORM`, and `URL` are scanned. These are the only ones that might contain external data. When set to **custom**, you can define which of the scopes will be scanned by Railo. By default, no scope is checked for external data (none).

This setting is just a default setting. You can use the attributes of the tag `<cfapplication>` or the properties of the `Application.cfc` in order to override this default. For example, to set the name and script protection specifically for your application, you can use the following `<cfapplication>` tag:

```
<cfapplication name="MyApplication"
scriptprotect="true">
```

For more information on XSS, see `http://en.wikipedia.org/wiki/Cross-Site_Scripting`.

Request timeout

The request timeout field allows you to set the maximum time a CFML page can run, before an error is thrown mentioning that the request ran too long. You can also define whether this setting can be overruled with a parameter in the URL called `RequestTimeout`. For example, you could set the maximum timeout of a page that you know will run a long time by calling it as follows:

```
http://localhost:8888/chapter_4/Listing4_4.cfm?RequestTimeout=50000
```

And this would change the request timeout of the page to `50` seconds (the time is measured in milliseconds).

Application listener

Applications in Railo Server can be defined with the use of a special file called `Application.cfc`. We shall see more on how this behaves in the next chapter, but for the moment, you can use this section to set which behaviors you want to support in your application.

An **Application Listener** is a special file or function that is run at different points during the request. You can set it to any of the following:

- **None**: This means Railo server will not look for specific files at the start and end of a request.
- **Classic (CFML < 7)**: This means Railo Server will look for a file called `Application.cfm` at the start of each request. If there is a file called `OnRequestEnd.cfm` in the same directory, it will include this template at the end of the request.
- **Modern**: This means Railo server will only look for a file called `Application.cfc` at the start of the request, not the `Application.cfm` or `OnRequestEnd.cfm`.
- **Mixed (CFML >= 7)**: This means Railo server will handle both the `Classic` and `Modern` patterns of template inclusion.

The next section for the Application Listener settings is called `mode`, and defines how Railo Server will look up the location of the `Application.cfc/.cfm` files.

- **Current**: This means Railo Server will only look for the `Application.cfc/Application.cfm` file in the current directory of the requested page.
- **Root**: This means Railo Server will only look for the `Application.cfc/Application.cfm` file in the web root of the site.
- **Current to Root**: This default setting will look for an `Application.cfc/Application.cfm` file starting from the current template path to the web root.

Output

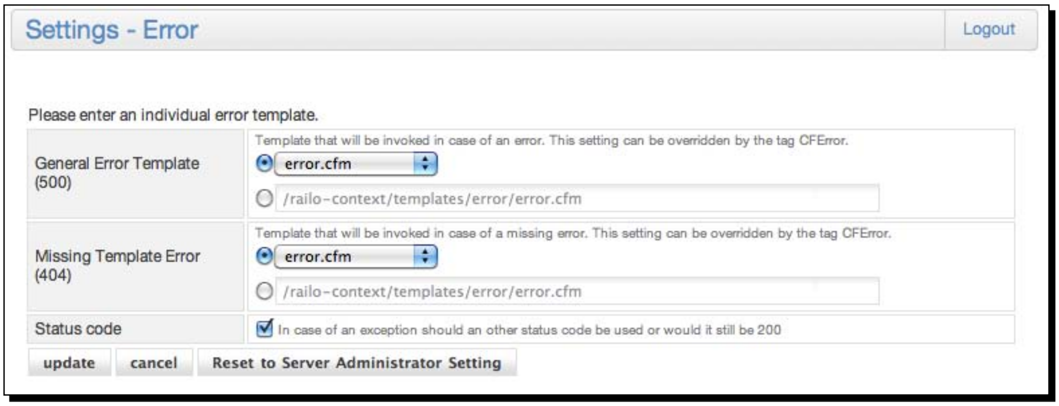
This section of the Web Administrator allows you to define special settings that give output to the user.

- **Whitespace management**: If this is set, Railo Server removes all extra whitespace (tabs, spaces, and carriage returns) from between the code that is output to the browser.
- **Output Railo Version**: If this is set, Railo Server returns the version of Railo you are running in the HTTP headers.
- **Suppress Content for CFC Remoting**: If this is set, Railo Server removes any content from inside a function call called remotely. Therefore, all that is returned is what the function returns, rather than any content inside the call.

Error

This section of the Web Administrator allows you to define how errors are handled. You can define templates for:

- **General Error Template**: This is invoked in case of a coding or server error. You can define your own custom error page.
- **Missing Template Error**: This template is called when you call for a `.cfm` or `.cfc` file that doesn't exist. You can define your own custom error template.
- **Status Code**: Normally, when a website has an error, an appropriate HTTP status code is returned. Here, you can set whether you will return this status code, or the status code `200`, which suggests that the request went okay (on an HTTP level).



Settings - Error

Please enter an individual error template.

General Error Template (500)	Template that will be invoked in case of an error. This setting can be overridden by the tag CFError. ● error.cfm ○ /railo-context/templates/error/error.cfm
Missing Template Error (404)	Template that will be invoked in case of a missing error. This setting can be overridden by the tag CFError. ● error.cfm ○ /railo-context/templates/error/error.cfm
Status code	☑ In case of an exception should an other status code be used or would it still be 200

Services

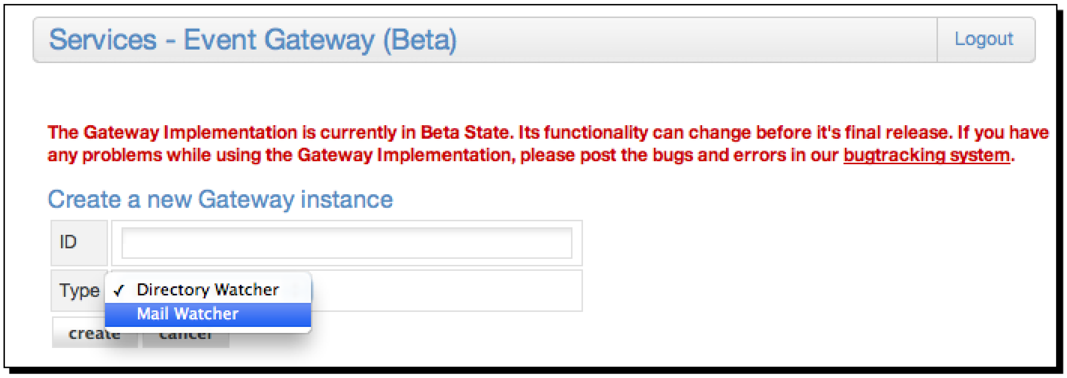
The **services** section of the Web Administrator allows you to define connections and settings to external systems, such as databases, caches, and e-mail servers. It also allows you to see running tasks and set up scheduled tasks.

Let's briefly cover some of the sections, since they will be covered later in the book in more detail.

Event Gateway

Event Gateway allows the Railo Server to handle requests or events from other systems outside a normal HTTP request. We are going to go into more detail about configuring and using **Event Gateway** in the next chapter.

In this section, you can choose from the built-in **Event Gateway**, such as the **Directory Watcher** and the **Mail Watcher**, as well as any others that might be installed.



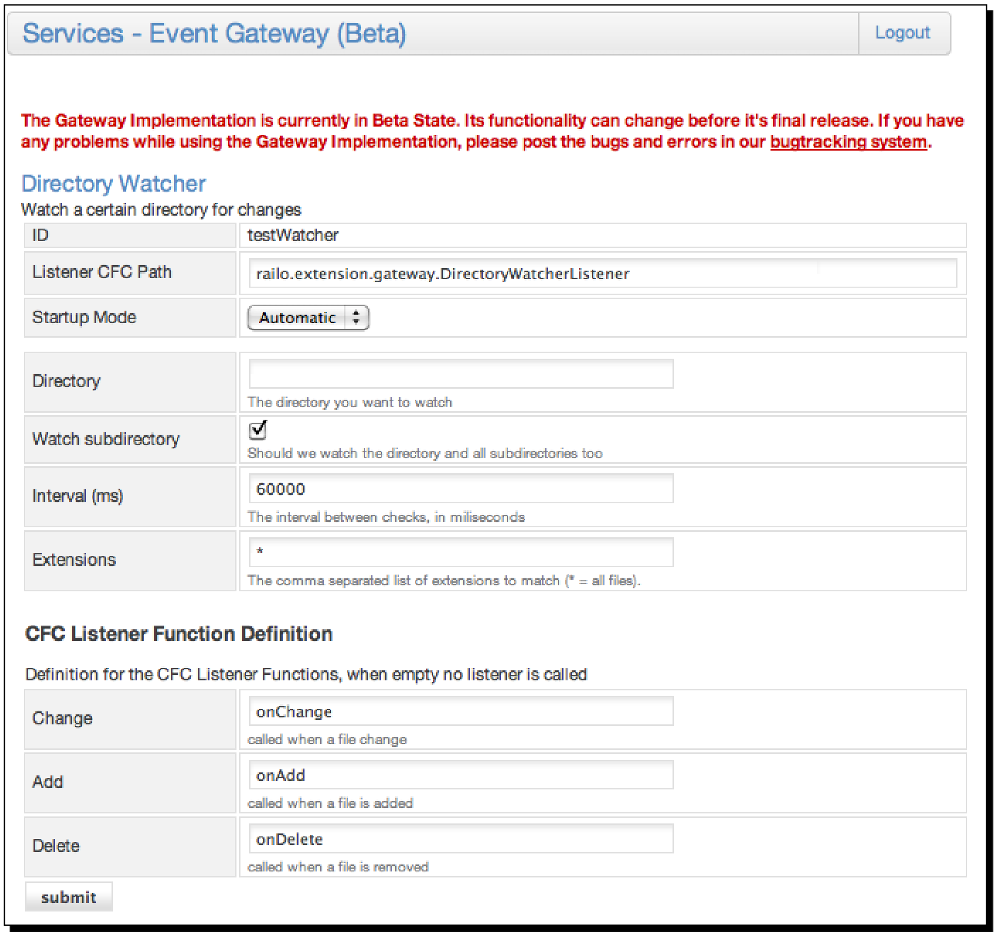
Services - Event Gateway (Beta)

The Gateway Implementation is currently in Beta State. Its functionality can change before it's final release. If you have any problems while using the Gateway Implementation, please post the bugs and errors in our bugtracking system.

Create a new Gateway instance

ID	
Type	✓ Directory Watcher

Once you have given the **Event Gateway** an **ID**, you will get a screen to configure it. As an example, here is the **Directory Watcher** configuration screen:



Services - Event Gateway (Beta)

The Gateway Implementation is currently in Beta State. Its functionality can change before it's final release. If you have any problems while using the Gateway Implementation, please post the bugs and errors in our bugtracking system.

Directory Watcher

Watch a certain directory for changes

ID	testWatcher
Listener CFC Path	railo.extension.gateway.DirectoryWatcherListener
Startup Mode	Automatic
Directory	The directory you want to watch
Watch subdirectory	☑ Should we watch the directory and all subdirectories too
Interval (ms)	60000 The interval between checks, in miliseconds
Extensions	* The comma separated list of extensions to match (* = all files).

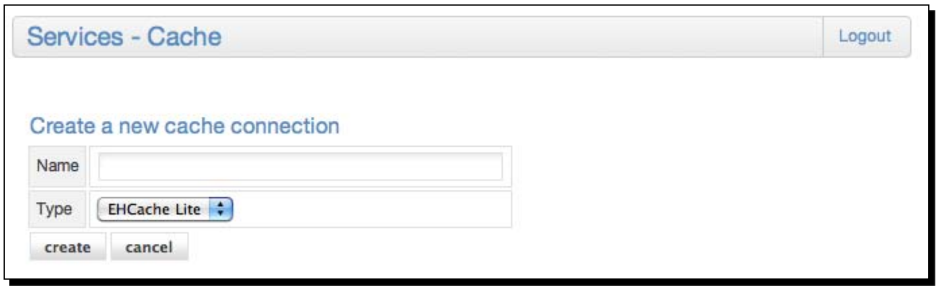
CFC Listener Function Definition

Definition for the CFC Listener Functions, when empty no listener is called

Change	onChange called when a file change
Add	onAdd called when a file is added
Delete	onDelete called when a file is removed

Cache

Railo Server provides a number of caches that you can use to save various variables to improve the speed of lookups, especially when getting frequently-used data. Built-in caches include **EHCache Lite** and **RamCache**. There are a number of functions that you can use to save objects into the cache and get them back again. This can vastly improve the speed of your application, rather than having to re-create the objects or save them in other ways.



We will look further into caches later on in the book. This section of the administrator will allow you to create caches, set properties of the cache, and to assign it as the default cache for various operations.

Datasource

The **Datasource** section of the Railo Web Administrator allows you to add connections to databases. By default, Railo Server comes with a range of connectors to different database servers, and you can even add your own JDBC drivers to connect to other database types.

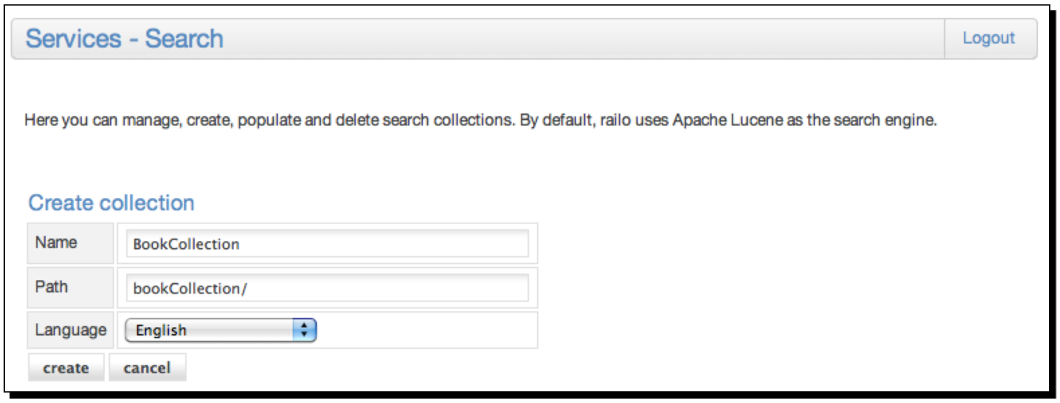
By default, Railo Server supports the following databases: DB2, Firebird, H2, HSQL DB, Microsoft SQL Server, MySQL, PostgreSQL, Sybase, Oracle, JDBC-ODBC Bridge (to connect to your ODBC datasources), and any other database that has a JDBC driver.

ORM

The **Object Relational Mapping** (**ORM**) section of the Web Administrator allows you to define settings on how Railo Server will interact with the included Hibernate ORM service. Railo Server provides a way for you to "persist" CFML Components to a database and relate them to each other. This means you don't have to do SQL queries to the database and you can just code your whole application using components that are saved to the database for you. We shall have a better look at this functionality in *Chapter 5*, *Developing Applications with Railo Server*.

Search

Railo Server contains a full text search engine. It uses the Lucene search engine from the Apache Foundation. The underlying engine can be swapped-out, since Railo Server uses an interface for the abstraction of the full text search. This enables you to add other search engines, if required, but your CFML code won't need to change.



Here you can manage, create, populate and delete search collections. By default, railo uses Apache Lucene as the search engine.

Create collection

Name	BookCollection
Path	bookCollection/
Language	English

In order to use the search functionality, we have to create a collection first. This can be done either within the Railo Web (or Server) Administrator or with the `<cfcollection>` tag.

You define a collection by giving it a name, a path where the collection index is stored, and the language that the collection is in.

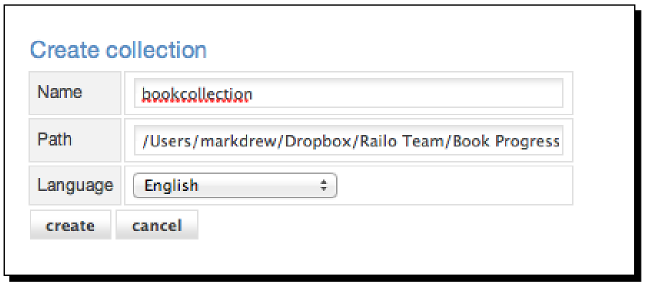
A collection is a group of items that Apache Lucene will allow you to search. These can be HTML files, PDF documents, Microsoft Word documents, and even web pages from other sites.

Time for action – creating a search collection

Let's create a collection and use the built-in search functionality to see it work.

1. We first need to create a directory to hold our collection; this doesn't have to be web-visible, but just for this example, we are going to create a folder in `<Railo Install Directory>/webroot` called `collections`.
2. Now, in the Railo Web Administrator, head to the **Search** section.
3. Enter `bookcollection` in the **Name** field for the **collection**.

4. Then, enter the full path to the `collection` folder in the **Path** field, and we should have something similar to the following screenshot:



Create collection

Name	bookcollection
Path	/Users/markdrew/Dropbox/Railo Team/Book Progress
Language	English

5. Click on **create**, and you will have a page listing the created `collection`.

6. We need something to search, so let's create a folder in the `<Railo Install Directory>/webroot` called `searchitems`.

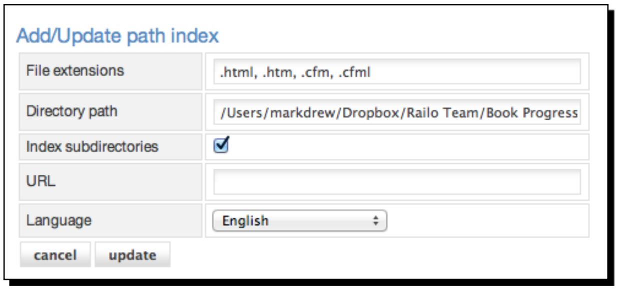
7. In this folder, let's add a page that we can now search. Let's write a file called `railorocks.html` and add the following to the file:

```
<!DOCTYPE html >
<html>
  <head>
    <title>A search item!</title>
  </head>
  <body>
    <h1>Railo Rocks!</h1>
  </body>
</html>
```

8. The `railorocks.html` is just a simple HTML file with the word `Railo` in it that we are going to search.

9. Let's head back to the Railo Web Administrator and click on the pencil icon next to the `bookcollection` we created before.

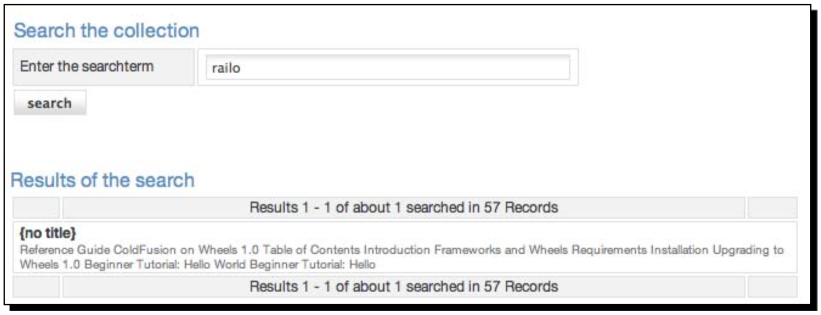
10. In the **Add/Update** path index section, add the full path to our `searchitems` folder, so it should look something like the following screenshot, and then click on **update**:



Add/Update path index

File extensions	.html, .htm, .cfm, .cfml
Directory path	/Users/markdrew/Dropbox/Railo Team/Book Progress
Index subdirectories	☑
URL	
Language	English

11. Now click on the pencil next to the `bookcollection` collection, and at the bottom, you can enter `Railo` in the **Enter the searchitem** textbox and click on **search**. You should now have an example result of the contents in that collection.



Search the collection

Enter the searchterm	railo

Results of the search

Results 1 - 1 of about 1 searched in 57 Records

{no title}
Reference Guide ColdFusion on Wheels 1.0 Table of Contents Introduction Frameworks and Wheels Requirements Installation Upgrading to Wheels 1.0 Beginner Tutorial: Hello World Beginner Tutorial: Hello

Results 1 - 1 of about 1 searched in 57 Records

What just happened?

Using the Railo Web Administrator, we added a new collection to the Apache Lucene engine. We then added a path, which we set to look through `.htm`, `.html`, `.cfm`, and `.cfml` files.

Railo Server will then maintain this collection, allowing you to add it to your site by using the `<cfsearch>` tag.

Have a go hero

Why not create a template with a form to search the collection? You could also add other types of files, such as PDF and Word Documents. To search this collection, you will need to use the `<cfsearch>` tag. Here is a quick example, but why not explore what else you can do?

```
<cfsearch collection="bookcollection" criteria="railo"
   name="findBooks">
<cfoutput query="findBooks">
  #score# - #title# - #summary#
</cfoutput>
```

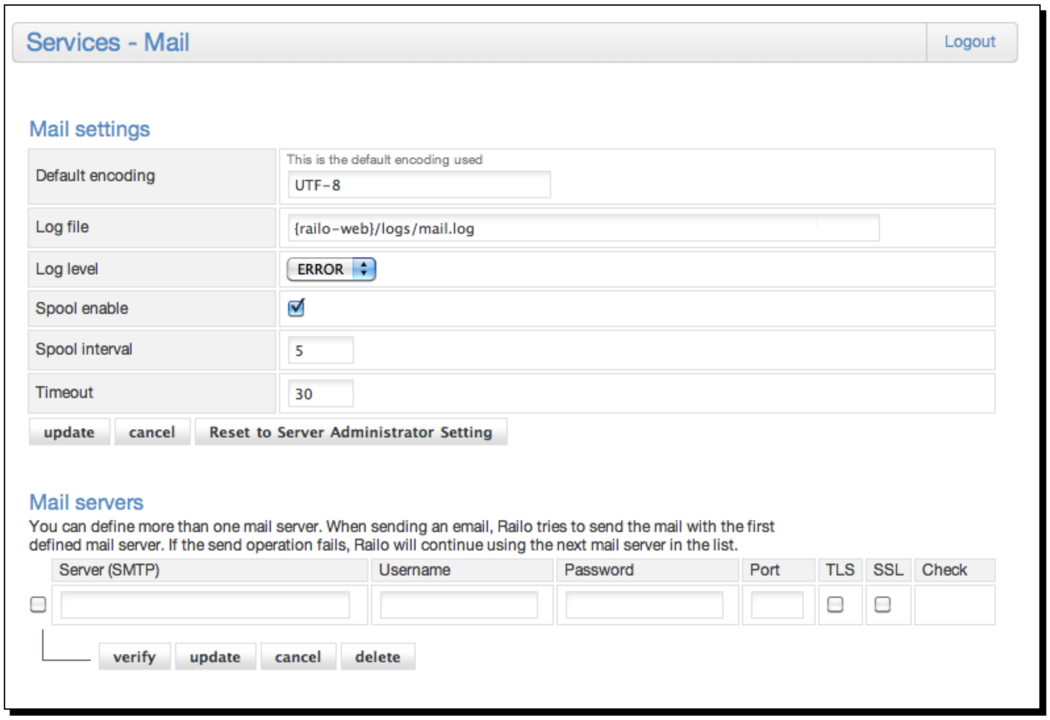
Mail

Most web applications, at some point, need to send an e-mail. For example, e-mails can be used to notify a user, to send error reports, or to even send out a newsletter. The **Mail** section of the Railo Web Administrator allows you to define global settings, such as the default encoding, the location of the log file, the kind of logs that will be kept, whether the spooler is enabled, the spool interval, and the timeout for sending mails.

You can also define SMTP servers that you will be using, the username, password, the port for that server, and whether it has been enabled. Then to send mail in your code, all you have to do is add the following code:

```
<cfmail from="me@mydomain.com" to="them@theirdomain.com"
subject="This is an email from Railo">
  It's so easy to send an email from Railo Server! You should try it!
</cfmail>
```

You can define a number of mail servers, and Railo Server will attempt each mail server in turn if one fails as a fall-back.



Services - Mail

Mail settings

Default encoding	This is the default encoding used UTF-8
Log file	{railo-web}/logs/mail.log
Log level	ERROR
Spool enable	☑
Spool interval	5
Timeout	30

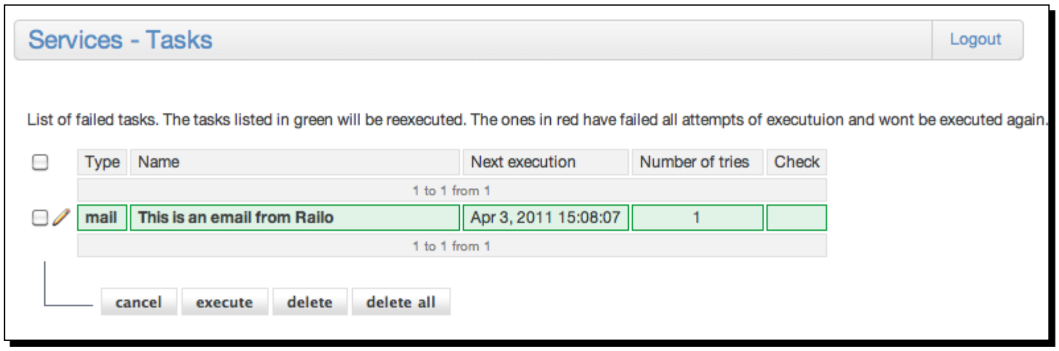
Mail servers

You can define more than one mail server. When sending an email, Railo tries to send the mail with the first defined mail server. If the send operation fails, Railo will continue using the next mail server in the list.

	Server (SMTP)	Username	Password	Port	TLS	SSL	Check
☐					☐	☐	

Tasks

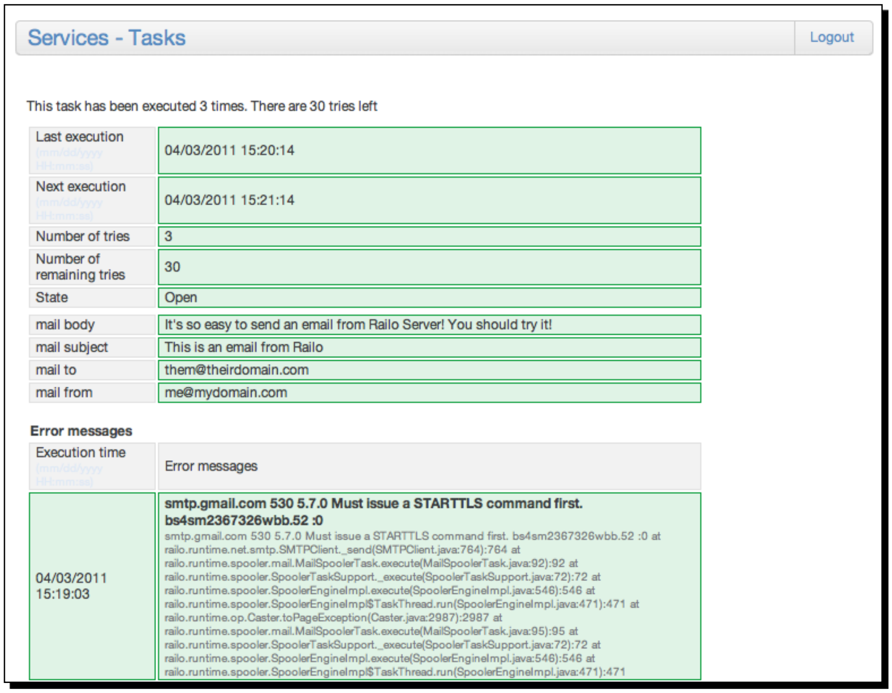
There are certain things that happen in the background with Railo Server, for example, sending an e-mail. This happens asynchronously, as it is put into a queue and sent later. This means your page will process normally and send the e-mail in the background. The problem with this is that the e-mail might end up not being sent for whatever reason. In the **Tasks** section, you can see a list of items that are being processed.



List of failed tasks. The tasks listed in green will be reexecuted. The ones in red have failed all attempts of executuion and wont be executed again.

□	Type	Name	Next execution	Number of tries	Check
	1 to 1 from 1				
□ ✎	mail	This is an email from Railo	Apr 3, 2011 15:08:07	1	
	1 to 1 from 1				

You can see that it will tell you how many times it has tried to send it, the type of task, and the name of the task in the list of tasks. By clicking on the **edit** button, we can get more details about why the task might still be active (or failed) in more detail:



This task has been executed 3 times. There are 30 tries left

Last execution (mm/dd/yyyy HH:mm:ss)	04/03/2011 15:20:14
Next execution (mm/dd/yyyy HH:mm:ss)	04/03/2011 15:21:14
Number of tries	3
Number of remaining tries	30
State	Open
mail body	It's so easy to send an email from Railo Server! You should try it!
mail subject	This is an email from Railo
mail to	them@theirdomain.com
mail from	me@mydomain.com

Error messages

Execution time (mm/dd/yyyy HH:mm:ss)	Error messages
04/03/2011 15:19:03	**smtp.gmail.com 530 5.7.0 Must issue a STARTTLS command first. bs4sm2367326wbb.52 :0** smtp.gmail.com 530 5.7.0 Must issue a STARTTLS command first. bs4sm2367326wbb.52 :0 at railo.runtime.net.smtp.SMTPClient._send(SMTPClient.java:764):764 at railo.runtime.spooler.mail.MailSpoolerTask.execute(MailSpoolerTask.java:92):92 at railo.runtime.spooler.SpoolerTaskSupport._execute(SpoolerTaskSupport.java:72):72 at railo.runtime.spooler.SpoolerEngineImpl.execute(SpoolerEngineImpl.java:546):546 at railo.runtime.spooler.SpoolerEngineImpl$TaskThread.run(SpoolerEngineImpl.java:471):471 at railo.runtime.op.Caster.toPageException(Caster.java:2987):2987 at railo.runtime.spooler.mail.MailSpoolerTask.execute(MailSpoolerTask.java:95):95 at railo.runtime.spooler.SpoolerTaskSupport._execute(SpoolerTaskSupport.java:72):72 at railo.runtime.spooler.SpoolerEngineImpl.execute(SpoolerEngineImpl.java:546):546 at railo.runtime.spooler.SpoolerEngineImpl$TaskThread.run(SpoolerEngineImpl.java:471):471

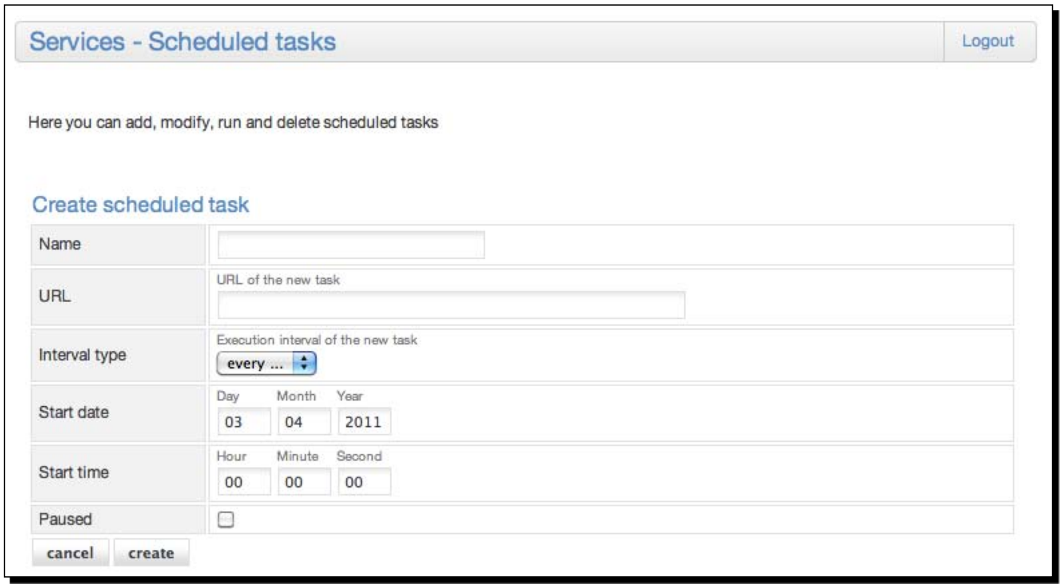
In this case, for example, we can see that the Gmail SMTP server returned the following error:

```
smtp.gmail.com 530 5.7.0 Must issue a STARTTLS command first.
```

The **Tasks** section is very good for debugging errors with asynchronous tasks.

Scheduled tasks

There are times when you want a process to happen on a regular basis. Of course, you could hire someone to hit a URL on your server every few hours, but it is easier to set up a scheduled task. In essence, a scheduled task will call a URL at defined times.



Services - Scheduled tasks

Here you can add, modify, run and delete scheduled tasks

Create scheduled task

Name	
URL	URL of the new task
Interval type	Execution interval of the new task every ...
Start date	Day 03 Month 04 Year 2011
Start time	Hour 00 Minute 00 Second 00
Paused	□

To create a scheduled task, all you need to do is enter:

- **Name**: The name you are going to give this scheduled task
- **URL**: The URL that will be called
- **Interval Type**: Whether you want to run this task once, daily, weekly, monthly, or at more specific intervals ("every...")
- **Start date**: When you want this task to start running
- **Start time**: At what time you want this task to start

Once you have saved this basic information, you will be able to add more detailed configuration information, for example, you will be able to set:

- The **Username** and **Password** if the **URL** is protected
- Any proxy settings, if you are using a proxy server to access the **URL**
- Whether logging should be enabled and where the log file will be stored
- Precisely when the execution should be run and until when

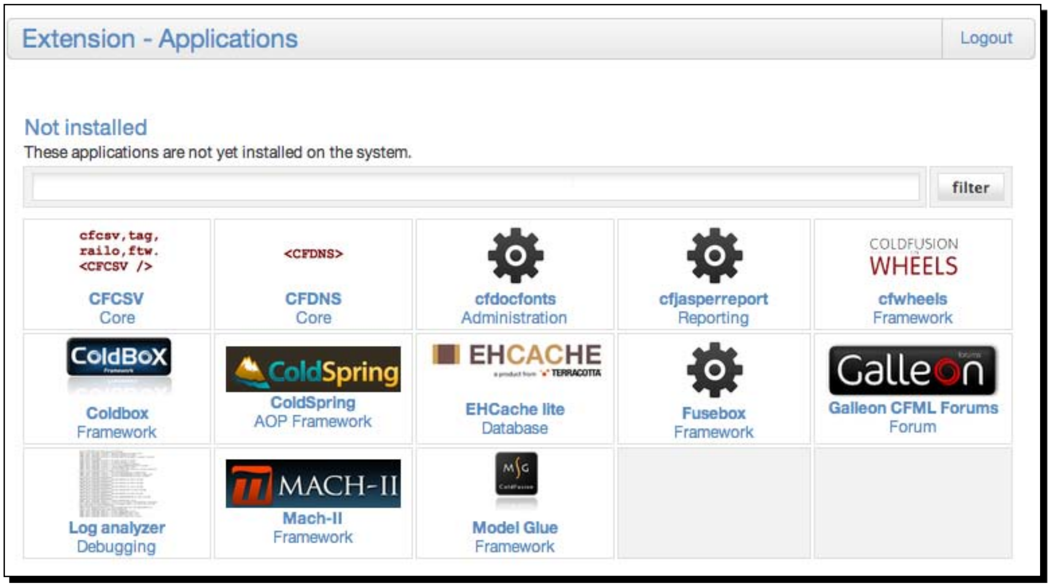
The task will then run until you either set it to paused, delete the task, or you set an end date for the task to stop executing.

Extension

The **Extension** section of the Railo Web (and Server) administrator allows you to install external applications and extensions directly into your web context. There are currently a number of applications (and growing) that can be installed, as well as different functional additions to the Railo Server that you can use, for example, different cache providers, resource providers, and tags.

Applications

The **Applications** section of the Railo Web Administrator shows applications that can be installed into your web context, for example, here is a list of the applications available:



Extension - Applications

Logout

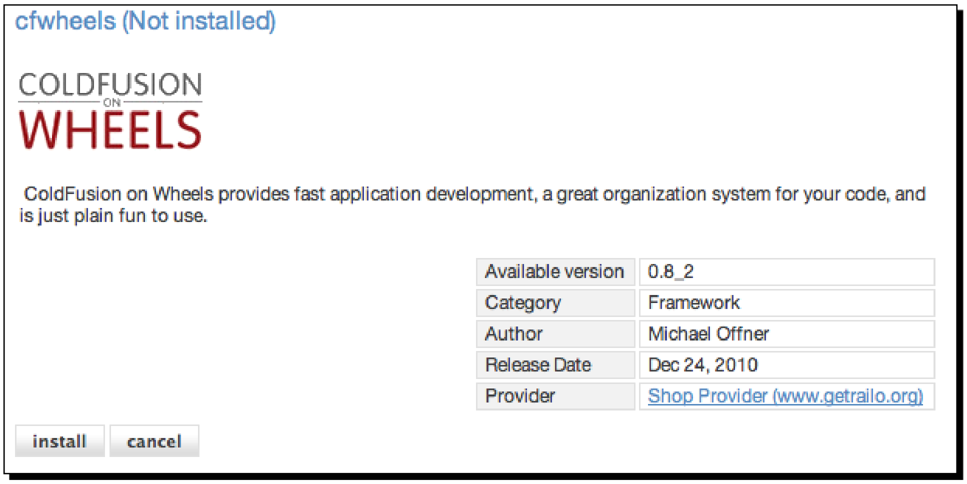
Not installed

These applications are not yet installed on the system.

filter

CFCSV Core	CFDNS Core	cfdocfonts Administration	cfjasperreport Reporting	cfwheels Framework
Coldbox Framework	ColdSpring AOP Framework	EHCache lite Database	Fusebox Framework	Galleon CFML Forums Forum
Log analyzer Debugging	Mach-II Framework	Model Glue Framework		

Selecting an application to install will give you some details about that application.

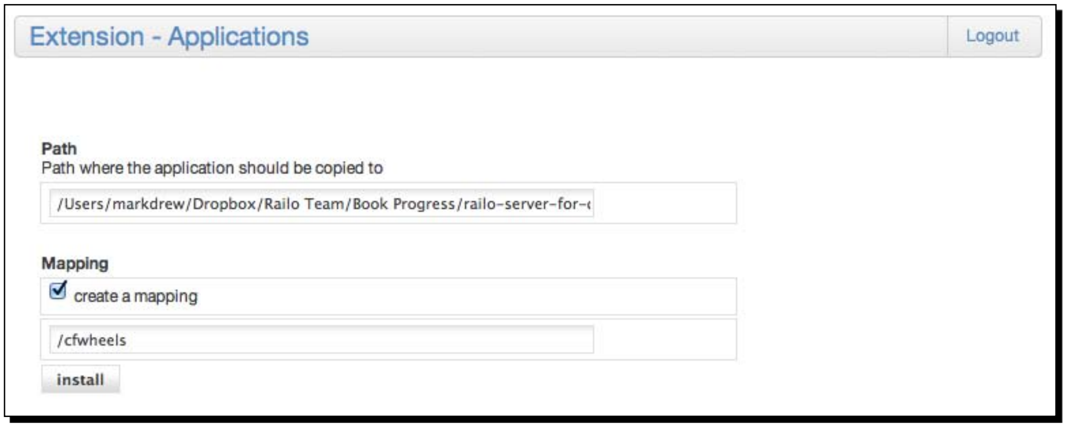


cfwheels (Not installed)

ColdFusion on Wheels provides fast application development, a great organization system for your code, and is just plain fun to use.

Available version	0.8_2
Category	Framework
Author	Michael Offner
Release Date	Dec 24, 2010
Provider	Shop Provider (www.getrailo.org)

Once you start the installation procedure by clicking on the **install** button, the extension will ask you specific instructions as to how that application needs to be installed. As an example, the application might ask you for the location to where it should be installed in, the datasource, and any other variable that needs to be set for it to work:



Extension - Applications

Path
Path where the application should be copied to

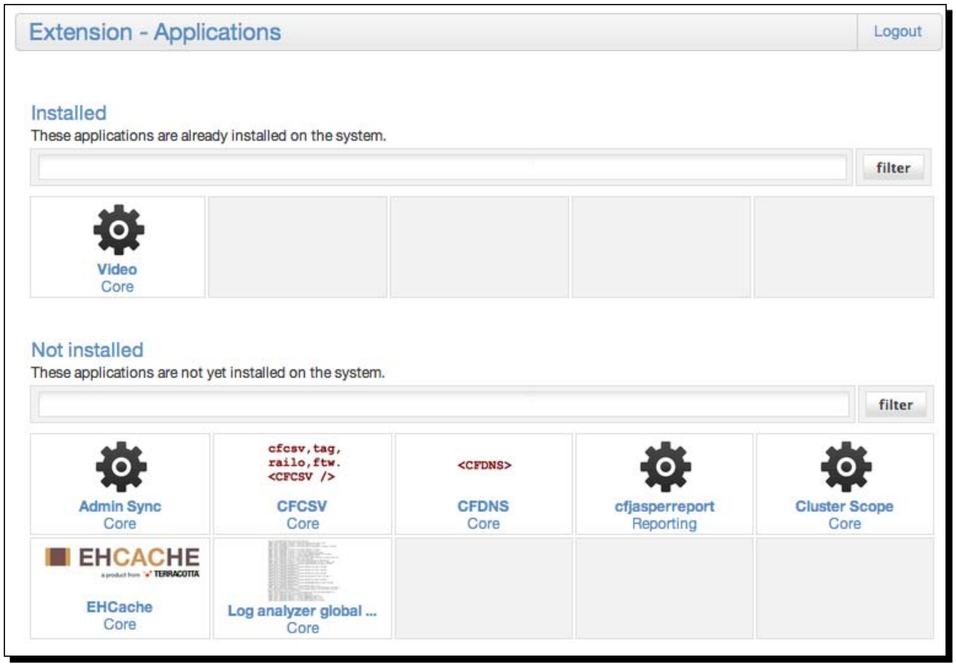
/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-c

Mapping

☑ create a mapping

/cfwheels

While you are in the applications screen, if you click on the **Server Administrator** tab at the top of the page, you will see the extensions that can be installed for a whole server:



Extension - Applications

Installed

These applications are already installed on the system.

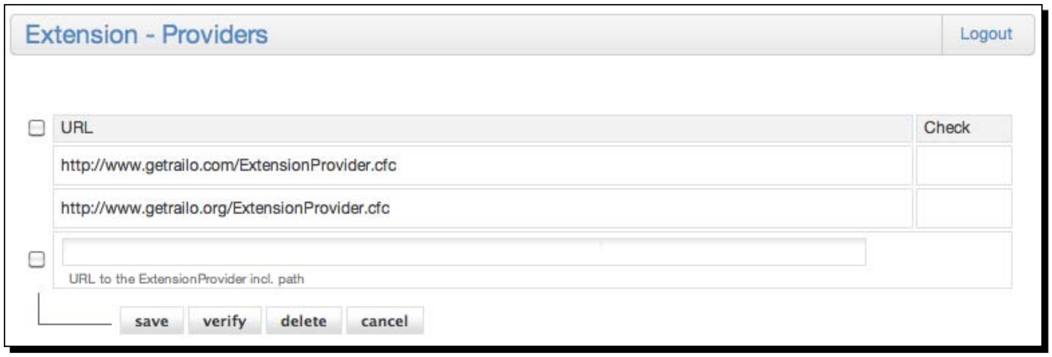
Not installed

These applications are not yet installed on the system.

The difference between the Server and Web Applications is that the Server applications usually provide functionality for all the contexts in a Railo Server, whereas the Applications in the Web Administrator usually install functionality just for that context.

Providers

Extensions and applications are provided from Railo Technologies directly, but you can also consume applications from other providers. The Providers screenshot, shown next, allows you to define other providers, including ones you wrote. We shall see more about this in *Chapter 9*, *Extending Railo Server*.

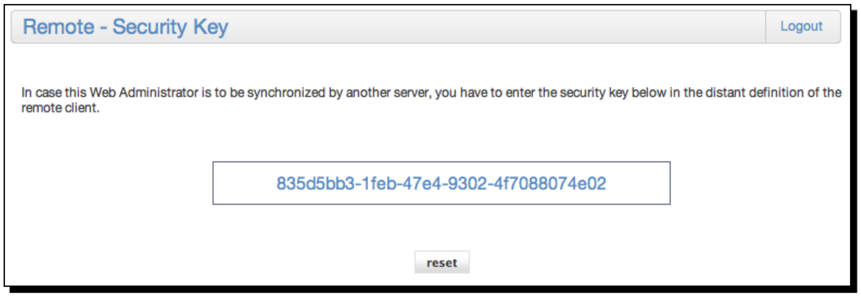


Extension - Providers

□	URL	Check
	http://www.getrailo.com/ExtensionProvider.cfc	
	http://www.getrailo.org/ExtensionProvider.cfc	
□	URL to the ExtensionProvider incl. path	

Remote

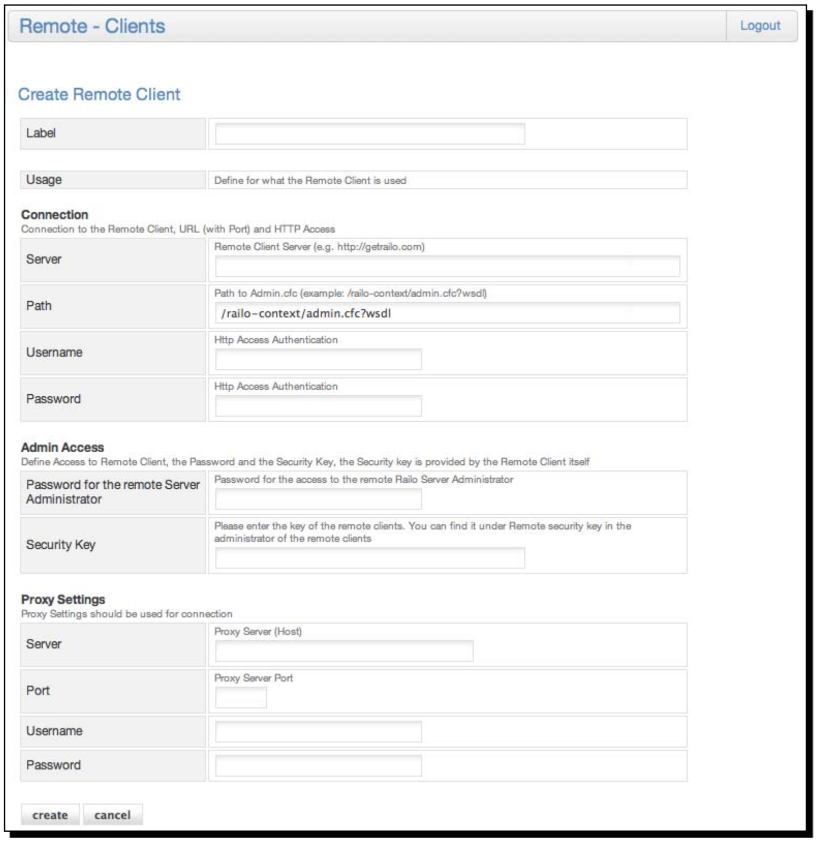
This section is used if you install the **Admin Sync** extension in the Server Administrator. The **Admin Sync** extension allows you to synchronize any changes you make to one server administrator across a number of other servers. To do this, you need to use the security key provided in the **Security Key** section:



In case this Web Administrator is to be synchronized by another server, you have to enter the security key below in the distant definition of the remote client.

835d5bb3-1feb-47e4-9302-4f7088074e02

You can then use that key when you are setting up another connection in the **Clients**:



Create Remote Client

Label	
Usage	Define for what the Remote Client is used

Connection
Connection to the Remote Client, URL (with Port) and HTTP Access

Server	Remote Client Server (e.g. http://getrailo.com)
Path	Path to Admin.cfc (example: /railo-context/admin.cfc?wsdl) /railo-context/admin.cfc?wsdl
Username	Http Access Authentication
Password	Http Access Authentication

Admin Access
Define Access to Remote Client, the Password and the Security Key, the Security key is provided by the Remote Client itself

Password for the remote Server Administrator	Password for the access to the remote Railo Server Administrator
Security Key	Please enter the key of the remote clients. You can find it under Remote security key in the administrator of the remote clients

Proxy Settings
Proxy Settings should be used for connection

Server	Proxy Server (Host)
Port	Proxy Server Port
Username	
Password	

More details on this can be found at `http://wiki.getrailo.com`, as it is outside the scope of this book.

Archives and resources

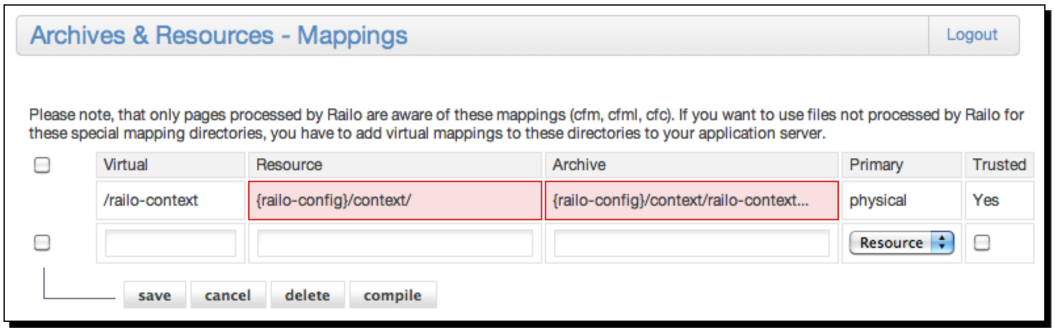
Railo Server has the ability to use a number of different filesystems as well as create aliases to code what you want to use. This section allows you to create those aliases, called **Mappings**, to securely access code that can be placed away from the webroot.

Mappings

Mappings are an easy way of accessing files and directories not available to an application. Mappings (like a virtual web server directory only for Railo), for example, allow you to access files that lie outside the webroot in your CFML code.

A mapping in Railo Server always starts with a slash `/`. If a `/` is at the beginning of a file location, Railo server checks any corresponding mapping and retrieves the file from the physical folder that the mapping points to. This is only valid for calls of `.cfm` or `.cfc` files through the browser, since these files are handled by Railo Server and mappings are only known to Railo Server. Other file types can be accessed through the `<cfinclude>` tag and `fileRead()` function (and in fact most of the `fileXXXX()` functions).

A web server knows nothing about Railo Server mappings. Therefore, it would be necessary to make Railo Server process all kinds of files, such as images and stylesheets, if you would like to use mappings for other file types.



Archives & Resources - Mappings

Please note, that only pages processed by Railo are aware of these mappings (cfm, cfml, cfc). If you want to use files not processed by Railo for these special mapping directories, you have to add virtual mappings to these directories to your application server.

□	Virtual	Resource	Archive	Primary	Trusted
	/railo-context	{railo-config}/context/	{railo-config}/context/railo-context...	physical	Yes
□				Resource	□

Within Railo Server, you can use mappings wherever you like. So a mapping works with a `<cffile>` tag and other file-related functions or tags.

Let's look at some examples

Time for action – creating mappings in our application

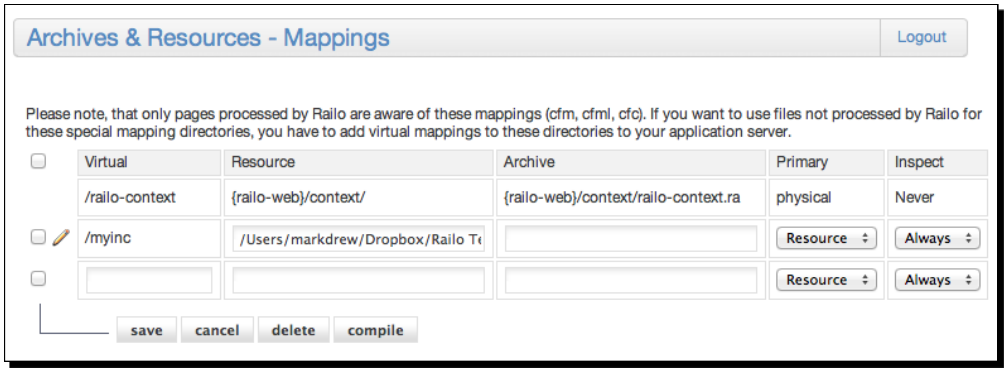
Let's create a mapping to include a file from our application.

1. In our `webroot`, let's create a folder called `includes`.

2. In the `includes` directory, add a template that will simply show the time; let's call it `inc.cfm` and add the following code:

   ```
   <cfoutput>#Now()#</cfoutput>
   ```

3. Then, in the Railo Web Administrator, head to the **Mappings** section, enter `/myinc` under the **Virtual** column entry box, and then enter the full path to the `includes` folder under the **Resource** column:



Archives & Resources - Mappings

Please note, that only pages processed by Railo are aware of these mappings (cfm, cfml, cfc). If you want to use files not processed by Railo for these special mapping directories, you have to add virtual mappings to these directories to your application server.

□	Virtual	Resource	Archive	Primary	Inspect
	/railo-context	{railo-web}/context/	{railo-web}/context/railo-context.ra	physical	Never
□	/myinc	/Users/markdrew/Dropbox/Railo Te		Resource	Always
□				Resource	Always

4. Once we have done that, click on the **save** button.

5. In our `webroot` folder, let's create another template this time called `mapping_test.cfm` and put the following code in it:

   ```
   <cfinclude template="/myinc/inc.cfm">
   ```

6. When we run our `mapping_test.cfm` template, by going to `http://localhost:8888/include_test.cfm`, we get the current time displayed:

   {ts '2011-09-19 09:47:24'}

What just happened?

By using a mapping, we were able to create an alias to a folder that we can use in our code. Railo Server is able to translate these mappings to specific files.

In Railo, mappings can be used wherever files are related in some way. So, for example, for the following tags we will follow the mappings:

- `<cfdirectory>`
- `<cffile>`
- `<cfinclude>`
- `<cfmodule>`

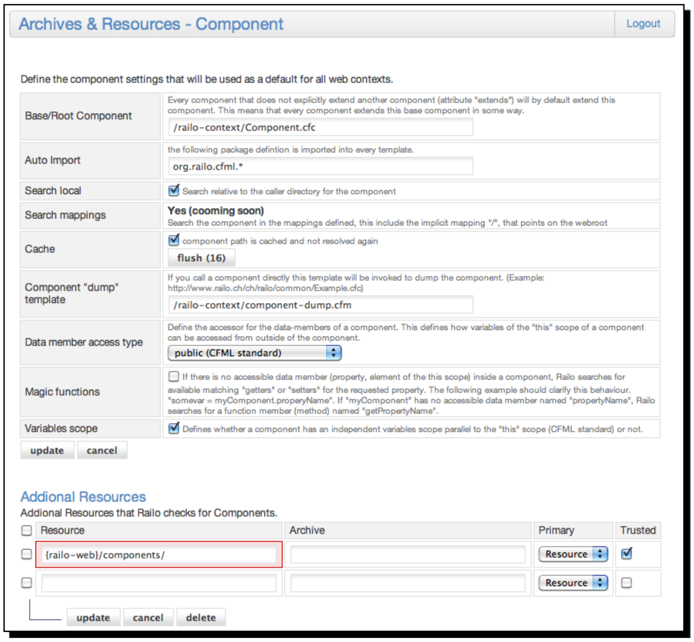
A mapping can also be marked as **Trusted** by selecting the **never** item in the drop-down under the **Inspect** column. This means that Railo Server will only check the contents of this mapping once, and then remember the content without re-checking whether the contents have changed or not. This is useful for sections of code that never change, and therefore, do not need to be re-compiled at any point.

We shall talk more in depth about resources in *Chapter 8*, *Resources and Mappings*.

Component

Components are used by Railo Server to provide self-contained modules of code. They are defined by creating a template with a `.cfc` extension (instead of the usual `.cfm` that we have seen so far).

They are analogous to Java classes, and have properties and functions that can act on those properties, but they are also more than that, as we shall see later in the book.



Define the component settings that will be used as a default for all web contexts.

Base/Root Component	Every component that does not explicitly extend another component (attribute "extends") will by default extend this component. This means that every component extends this base component in some way. /railo-context/Component.cfc
Auto Import	the following package defintion is imported into every template. org.railo.cfml.*
Search local	☑ Search relative to the caller directory for the component
Search mappings	**Yes (cooming soon)** Search the component in the mappings defined, this include the implicit mapping "/", that points on the webroot
Cache	☑ component path is cached and not resolved again flush (16)
Component "dump" template	If you call a component directly this template will be invoked to dump the component. (Example: http://www.railo.ch/ch/railo/common/Example.cfc) /railo-context/component-dump.cfm
Data member access type	Define the accessor for the data-members of a component. This defines how variables of the "this" scope of a component can be accessed from outside of the component. public (CFML standard)
Magic functions	☐ If there is no accessible data member (property, element of the this scope) inside a component, Railo searches for available matching "getters" or "setters" for the requested property. The following example should clarify this behaviour. "somevar = myComponent.properyName". If "myComponent" has no accessible data member named "propertyName", Railo searches for a function member (method) named "getPropertyName".
Variables scope	☑ Defines whether a component has an independent variables scope parallel to the "this" scope (CFML standard) or not.

Addional Resources

Addional Resources that Railo checks for Components.

☐	Resource	Archive	Primary	Trusted
☐	{railo-web}/components/		Resource	☑
☐			Resource	☐

This section allows you to define the defaults on how components are handled.

Base/Root Component

This setting defines the base component that every component will extend by default. (unless you use the `extends=""` attribute in the `<cfcomponent> tag`).

Auto import

There are built-in components available to every `.cfm` or `.cfc` template. These components are provided by default by Railo Server. We shall see more of these in Chapter 6, *Advanced Functionality*. But, in effect, it means that you can call Railo Server's built-in components as:

```
<cfset myFeed = new Feed()>
  Instead of using the full path:
<cfset myFeed = new org.railo.cfml.Feed()>
```

Search local

This will start the search for components in the directory relative to where you are calling the component from.

Cache

Looking up components can be a processor-intensive task. Therefore, once Railo Server has found a component, the location or path to that component is cached, speeding up subsequent calls to create that component.

Component "dump" template

Normally, when you call a component directly through the URL, you will get a dump of the properties and functions of that component. This is defined by the `/railo-context/component-dump.cfm` template. If you would like to do something different, such as provide documentation for that component when it is called, you can modify the template used to display a component.

Data member access type

Properties within a component are normally stored in the `THIS` scope and by default are `public`, that is, you can access and modify them externally. By changing the setting here, you can change this behavior and make them:

- `private`: The properties are only available from within the component itself
- `package`: The properties are only visible to other templates that are in the same folder
- `public`: The properties are visible to all the templates in the application
- `remote`: The properties are visible to remote (web service) calls to this component

Magic functions

Magic functions allow you to specify functions that will be called when you call or set a property of a component.

Let's look at an example to see how they work.

Time for action – using magic functions

1. Let's create a component that we are going to access. In our `webroot` folder, let's create a template called `PrivateComponent.cfc` and add the following code:

```
<cfcomponent>
  <cfset variables.name = "Test Value">
</cfcomponent>
```

2. This code has defined a component. Since the `VARIABLES` scope is `private`, by default, the variable name is also protected from being read directly.

3. Now let's create a template to call this component and see what happens when we call the `VARIABLES.name` variable. In the `webroot` folder, create a template called `magic.cfm` with the following code:

```
<cfset myComponent = new PrivateComponent()>
<cfset theName = myComponent.name>
<cfoutput>#theName#</cfoutput>
```

4. When we run the `magic.cfm` template, by going to `http://localhost:8888/magic.cfm`, we get the following error:

   Component [PrivateComponent] has no acessible Member with name [NAME]

5. That makes sense; the `VARIABLES` scope is a private scope. Let's enable magic functions. In the Railo Web Administrator, go to the **Components** section and tick the **Magic Functions** checkbox (if it hasn't been ticked already, of course) and click on the **update** button.

6. Now that we have enabled them, we can add a function to our component that will be called. Let's modify our `PrivateComponent.cfc` template to look like the following:

```
<cfcomponent>
  <cfset variables.name = "Test Value">
  <cffunction name="getName">
    <cfreturn variables.name>
  </cffunction>
</cfcomponent>
```

7. Now, let's re-run our `magic.cfm` template. We see that instead of getting an error, we get:

   Test Value

What just happened?

When we enable **Magic Functions** for our components, Railo Server sees that we are calling a property in a component, translates that to `get<Property Name>`, and runs the matching function in the component (if the function exists of course). Also, if you were setting a property in a component, it would call the `set<Property Name>`.

This is very handy if you have code that has, for example, been using key value structures and you want to replace that structure with a component to enable it to have more functionality.

Additional resources

The **Additional resources** section of the Component section works in the same way as mappings do, but it allows you to define overall additional locations to check for components.

Custom tags

We are going to explore custom tags in more detail in Chapter 9, *Extending Railo Server*. Custom tags are files written in CFML that you can call by name. So, for example, if we had a file called `hello.cfm` with the following content:

```
Hello There!
```

We could call it as follows from another template:

```
    <cf_hello>
```

And it would display **Hello There!**

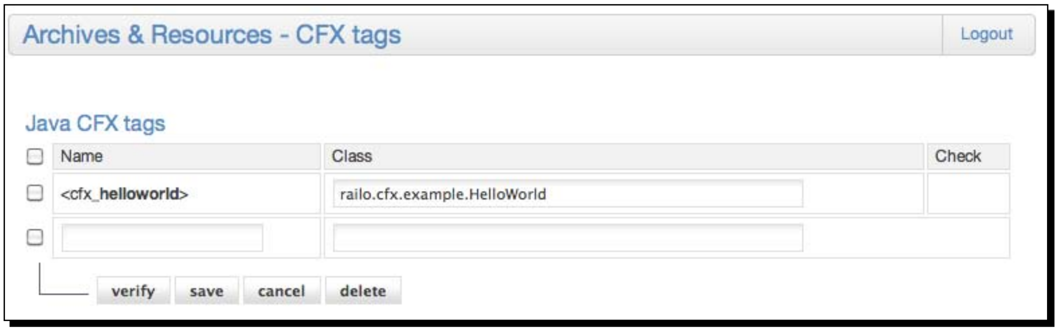
The **Custom tags** section allows you to define how Railo Server looks for custom tags, how it caches them, and what extensions you can use.

As an addition, you can also define custom tag mappings the same way that mappings are defined, and they are searched in addition to the normal mappings.

Setting	Description
Search Subdirectories	Tells Railo Server to search in the subdirectories for custom tags.
Search Local	Tells Railo Server to look in the current folder that we are calling the custom tags from.
Cache	To cache a path of a custom tag once it has been successfully called.
Extensions	Which extensions you can use for custom tags. By default, you can create component-based custom tags (`.cfc`) as well as simple template-based components.
Resources	Paths that Railo Server will search for custom tags.

CFX tags

In Railo Server, you can not only use custom tags written in CFML, but you can also have custom tags written in Java. There are a number of open source and commercial CFX tags developed, and by adding a library to your `<railo install>/lib/` directory, you can then define the custom tag in this section by giving it a name and a Java class to call.



Archives & Resources - CFX tags

Java CFX tags

□	Name	Class	Check
□	<cfx_**helloworld**>	railo.cfx.example.HelloWorld	
□			

Development

The debugging section allows you to turn on debugging for your context. If you set **Enable debugging** to **Yes**, when you call a page in Railo Server, you will get debug information, such as which components and templates have been called, how many times they have been called, and how long each template took to process.

Showing debugging output is very useful to see what templates you are calling, what queries are run, as well as other useful information.

Time for action – setting the debug template

Let's create a file to see what debug output we can get from Railo Server. We are going to add a query and include another file.

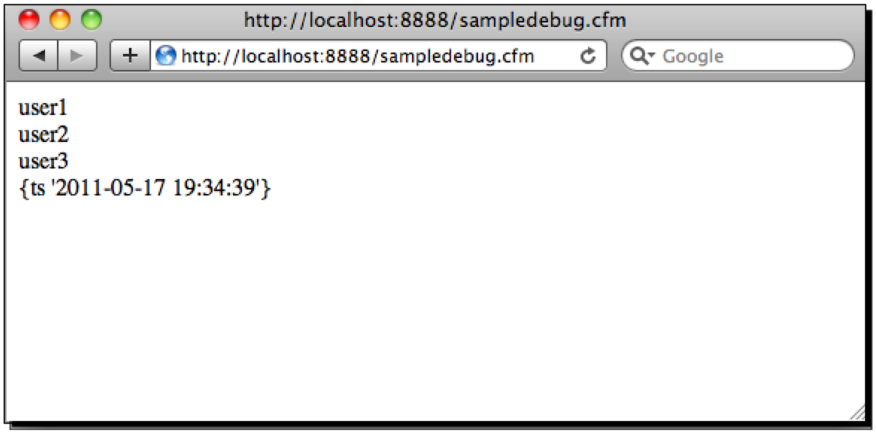
1. Under `<Railo Server Install>/webroot`, let's create a file called `example_debug.cfm`.
2. Using the datasource we created earlier, we are going to create a template so that we can display debugging. Let's first create a template that we are going to include in `<Railo Install Directory>/webroot/` called `testinclude.cfm`, and put the following code in it:

```
<cfoutput>#Now()#</cfoutput>
```

3. Now let's create a template that we shall debug; let's call it `sampledebug.cfm` and put it in the `webroot` too.
4. In this template, let's perform some actions, so that we can see some interesting output. Let's put the following code there:

```
<cfquery name="getUsers" datasource="railobook">
  SELECT * FROM Users
</cfquery>
<cfoutput query="getUsers">
  #Username# <br>
</cfoutput>
<cfinclude template="testinclude.cfm">
```

5. If we run this template currently, the output should look something like this:

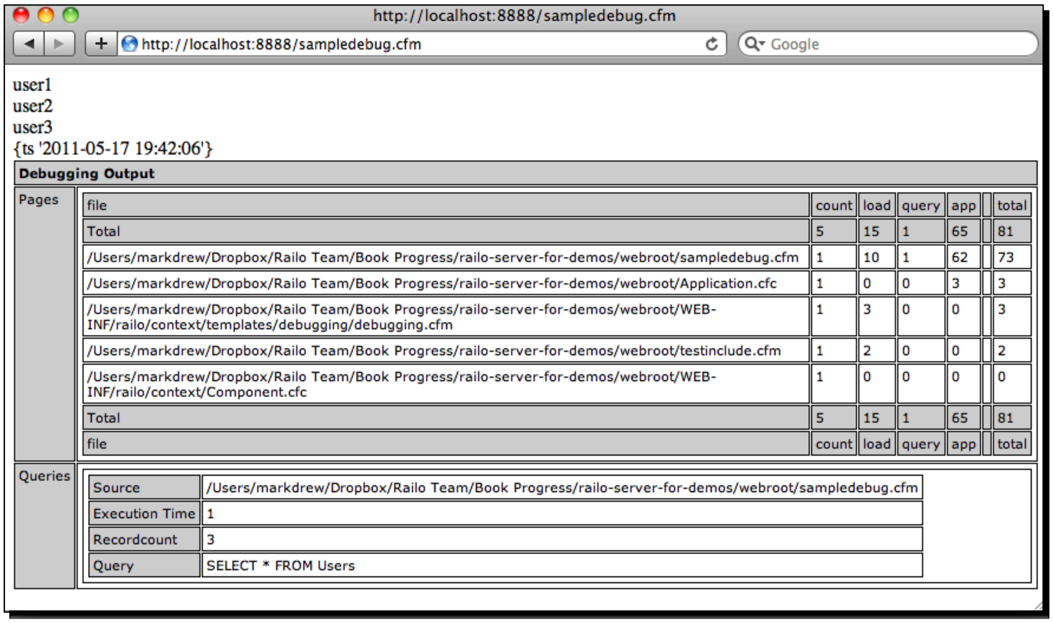

http://localhost:8888/sampledebug.cfm
http://localhost:8888/sampledebug.cfm
Google

```
user1
user2
user3
{ts '2011-05-17 19:34:39'}
```

6. As you can see, there is no debugging output being displayed. Let's change that by going to the Railo Web Administrator `http://localhost:8888/railo-context/admin/web.cfm` and then clicking on the **Debugging** link under **Development**.

7. To enable debugging, change the **Enable Debugging** drop-down to say **Yes** rather than the default setting, which should be **Server Administrator Value** (**No**), and click on **update**.

8. When we refresh our `sampledebug.cfm` template in the browser, we will now see the debugging output that Railo Server provides:



http://localhost:8888/sampledebug.cfm
http://localhost:8888/sampledebug.cfm
Google

```
user1
user2
user3
{ts '2011-05-17 19:42:06'}
```

Debugging Output

Pages

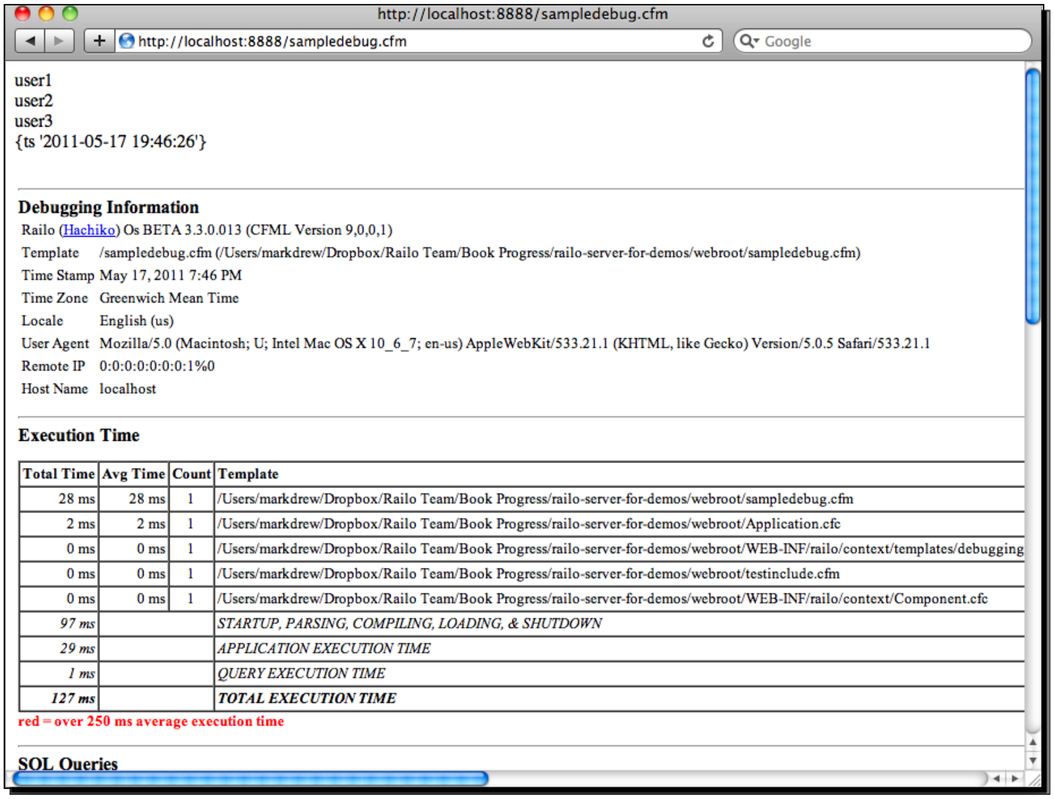
file	count	load	query	app		total
Total	5	15	1	65		81
/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/sampledebug.cfm	1	10	1	62		73
/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/Application.cfc	1	0	0	3		3
/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/WEB-INF/railo/context/templates/debugging/debugging.cfm	1	3	0	0		3
/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/testinclude.cfm	1	2	0	0		2
/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/WEB-INF/railo/context/Component.cfc	1	0	0	0		0
Total	5	15	1	65		81
file	count	load	query	app		total

Queries

Source	/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/sampledebug.cfm
Execution Time	1
Recordcount	3
Query	SELECT * FROM Users

9. It displays all the files that were called; including the base components and the debug tag itself. It also shows the queries that we run on this page.

10. We can change the debug output by going back to the **Debug** screen in the Railo Web Administrator and selecting a different debug template from the drop-down, such as `debugging-neo.cfm`, which will give us a different debugging output:



http://localhost:8888/sampledebug.cfm
http://localhost:8888/sampledebug.cfm
Google

user1
user2
user3
{ts '2011-05-17 19:46:26'}

Debugging Information

Railo (Hachiko) Os BETA 3.3.0.013 (CFML Version 9,0,0,1)	
Template	/sampledebug.cfm (/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/sampledebug.cfm)
Time Stamp	May 17, 2011 7:46 PM
Time Zone	Greenwich Mean Time
Locale	English (us)
User Agent	Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us) AppleWebKit/533.21.1 (KHTML, like Gecko) Version/5.0.5 Safari/533.21.1
Remote IP	0:0:0:0:0:0:0:1%0
Host Name	localhost

Execution Time

Total Time	Avg Time	Count	Template
28 ms	28 ms	1	/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/sampledebug.cfm
2 ms	2 ms	1	/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/Application.cfc
0 ms	0 ms	1	/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/WEB-INF/railo/context/templates/debugging
0 ms	0 ms	1	/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/testinclude.cfm
0 ms	0 ms	1	/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/WEB-INF/railo/context/Component.cfc
97 ms			*STARTUP, PARSING, COMPILING, LOADING, & SHUTDOWN*
29 ms			*APPLICATION EXECUTION TIME*
1 ms			*QUERY EXECUTION TIME*
127 ms			***TOTAL EXECUTION TIME***

red = over 250 ms average execution time

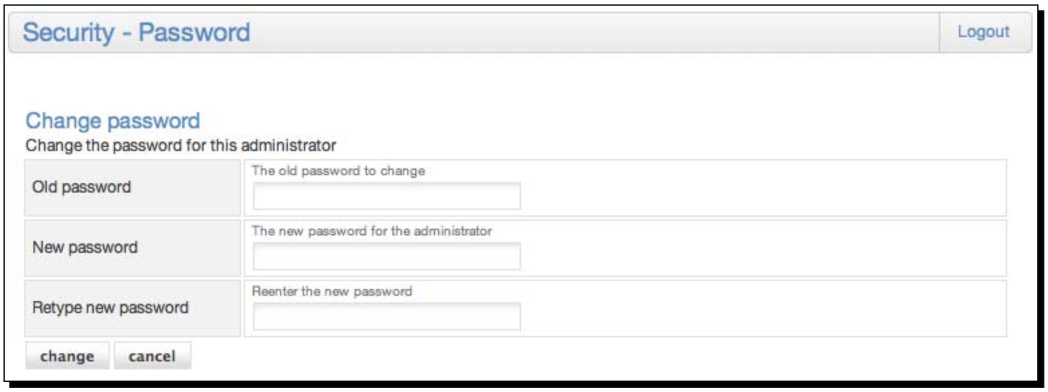
SQL Queries

What just happened?

By enabling debugging in the administrator, we are able to see exactly what our code is doing. It displays information about resources, scopes, components, and templates that we have included. If you want to, you can even create your own debugging template that would only show under certain conditions, such as a request from a specific URL.

Security

The **Password** section under **Security** allows you to change this context's password. Just enter your current password, your new password, and retype your new password. The next time you log into the Web Administrator, you will use your new password.



Security - Password

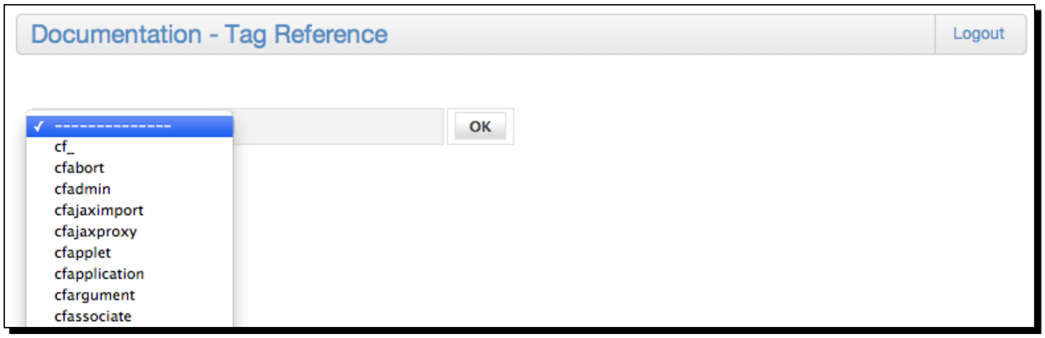
Change password

Change the password for this administrator

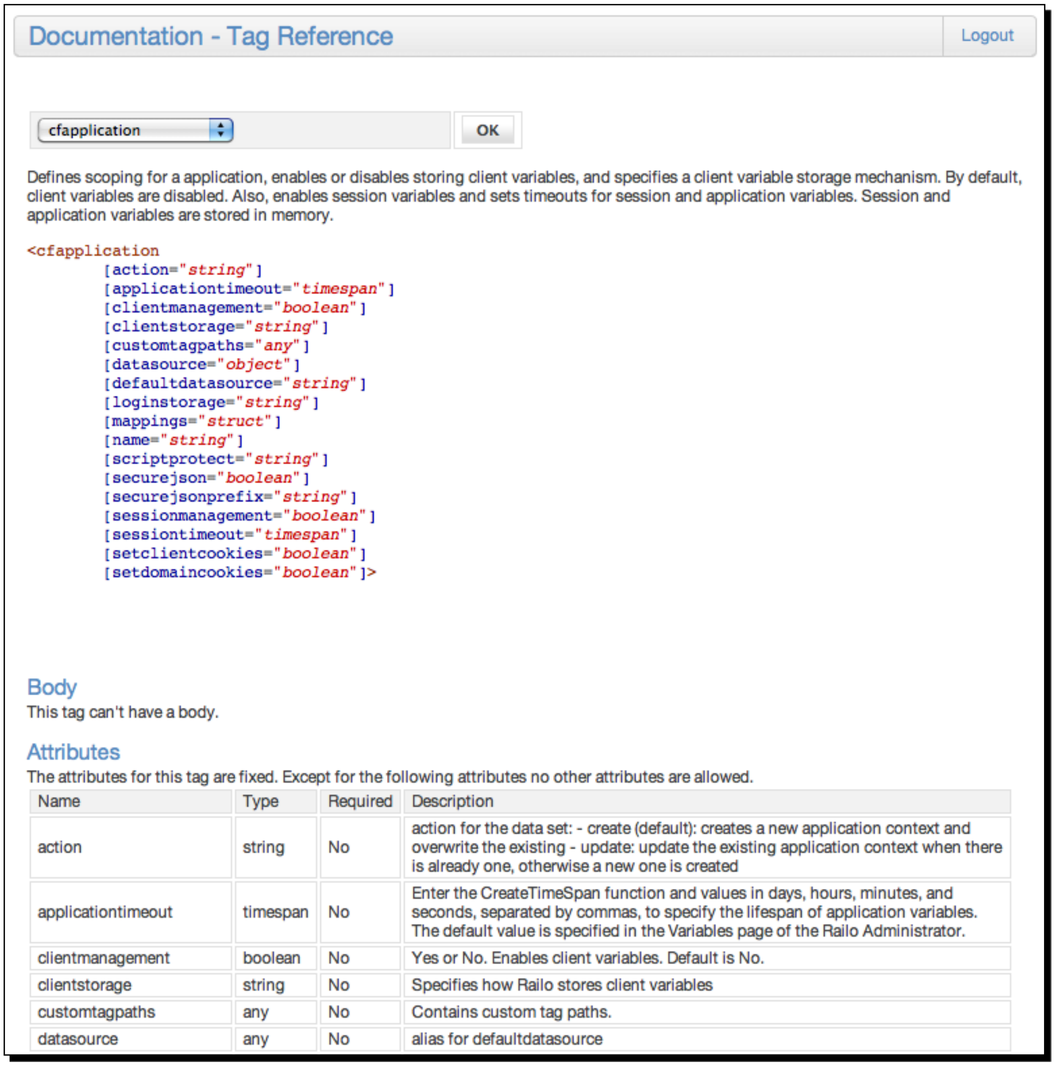
Old password	The old password to change
New password	The new password for the administrator
Retype new password	Reenter the new password

Documentation

Under the **Documentation** section, you will find some quick ways to look up **Tag** and **Function** references. They both work the same way, and you just need to select the tag or function and it will display a summary of its function and attributes/arguments that are valid for that tag.



Documentation - Tag Reference



Defines scoping for a application, enables or disables storing client variables, and specifies a client variable storage mechanism. By default, client variables are disabled. Also, enables session variables and sets timeouts for session and application variables. Session and application variables are stored in memory.

```
<cfapplication
        [action="string"]
        [applicationtimeout="timespan"]
        [clientmanagement="boolean"]
        [clientstorage="string"]
        [customtagpaths="any"]
        [datasource="object"]
        [defaultdatasource="string"]
        [loginstorage="string"]
        [mappings="struct"]
        [name="string"]
        [scriptprotect="string"]
        [securejson="boolean"]
        [securejsonprefix="string"]
        [sessionmanagement="boolean"]
        [sessiontimeout="timespan"]
        [setclientcookies="boolean"]
        [setdomaincookies="boolean"]>
```

Body

This tag can't have a body.

Attributes

The attributes for this tag are fixed. Except for the following attributes no other attributes are allowed.

Name	Type	Required	Description
action	string	No	action for the data set: - create (default): creates a new application context and overwrite the existing - update: update the existing application context when there is already one, otherwise a new one is created
applicationtimeout	timespan	No	Enter the CreateTimeSpan function and values in days, hours, minutes, and seconds, separated by commas, to specify the lifespan of application variables. The default value is specified in the Variables page of the Railo Administrator.
clientmanagement	boolean	No	Yes or No. Enables client variables. Default is No.
clientstorage	string	No	Specifies how Railo stores client variables
customtagpaths	any	No	Contains custom tag paths.
datasource	any	No	alias for defaultdatasource

Summary

We covered a lot of ground in this chapter. You should now have a good idea of:

- The Server and Web contexts available to your application
    - The Server context is a part of Railo Server that is used to define settings across web contexts
    - The Web context(s) are instances of Railo Server that allow administrators to manage their own context independently of other Web Contexts
- The settings available to change the time, output, performance, and internationalization of your server and web context
- How to set up different services, such as databases and datasources, caches, adding debug information to your requests, and setting up a connection to mail servers
- How to extend your server and web context with different applications available from different providers
- How to call templates and components through different mappings that are outside the web root

This chapter should have given you a great overview of configuring and customizing your server to your needs.

In the next chapter, we shall start using some of the services and archives and get to use some of the settings that we have seen in this chapter.

5

Developing Applications with Railo Server

Now that we have a good handle on the settings that are available in the web administrator for a context, let's turn our attention to the **Application** *Lifecycle and how we can use that to develop applications. In this chapter, we will have a look at:*

- Using the `Application.cfc` to manage the Application Lifecycle
- Using components to interact with our database using ORM
- Using the various caching techniques in Railo Server

By the end of the chapter, you should have a good understanding of what the `Application.cfc` file does, be able to interact with your database using components, and finally, be able to see how caching affects your application's performance.

Railo applications

Applications in Railo Server are defined as a number of templates working together. All the templates can share certain resources and scopes, such as the "APPLICATION" scope and interact with each other, for example, storing session information, setting and reading cookies from a user, sharing a datasource, and so on.

We have already seen in the Railo Web Administrator that we can set certain defaults for applications, such as the application and session timeouts, and so on, but what if we wanted to have more control than that?

Time for action – building the simplest application

Imagine that we are creating an application and all that it has to do is remember a user's name. The whole application will only consist of a couple of files – a form where a user can enter their name, and another page that will display their name. Then, we'll go ahead and make our application, it will have to remember these names without having the user submit it again.

1. Under the `webroot` in `<Railo Install Directory>/webroot`, create a sub-folder, where we are going to store our templates, called `HelloApp`.

2. Create a file in the `HelloApp` directory called `index.cfm` with the following code:

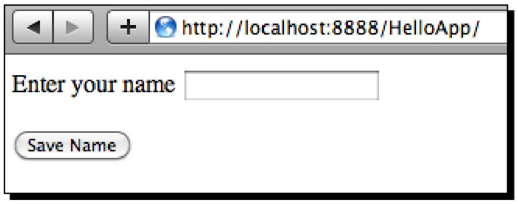
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Remember My Name</title>
  </head>
  <body>
    <form action="savename.cfm" method="post">
      <label>Enter your name</label>
      <input type="text" name="name" value="">
      <p><input type="submit" value="Save Name"></p>
    </form>
  </body>
</html>
```

3. Create another template called `savename.cfm` with the following code:

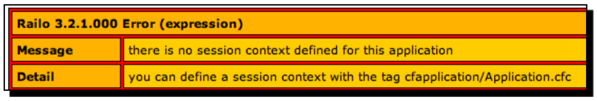
```
<!DOCTYPE html>
<cfparam name="FORM.name" default="">
<html lang="en">
  <head>
    <title>Saved your name</title>
  </head>
  <body>
    <cfset SESSION.name = FORM.name>Welcome
    <cfoutput>#SESSION.name#</cfoutput>!
  </body>
</html>
```

In this code, we have used the `<cfparam>` to set the `FORM.name` posted variable to a default value; this way, this template won't break if you don't send anything. Then we set the variable `SESSION.name` to the value we sent in the form by getting the variable from `FORM.name`. Finally, we output what is stored in the `SESSION.name` and display it in a friendly way.

4. Now go to `http://localhost:8888/HelloApp/` and see our newly created form, which will look like the following screenshot:



- Enter your name and submit the form
- Oh no! We get an error:



Railo 3.2.1.000 Error (expression)	
Message	there is no session context defined for this application
Detail	you can define a session context with the tag cfapplication/Application.cfc

What just happened?

We created a simple form to submit our name to another template, but we got an error. The error actually explains what is going on. We tried to set a variable to the `SESSION` scope, but we haven't enabled sessions in our application; in fact, we don't even have an application defined yet. There are just two templates in a folder. Let's fix this now.

Time for action – defining the application

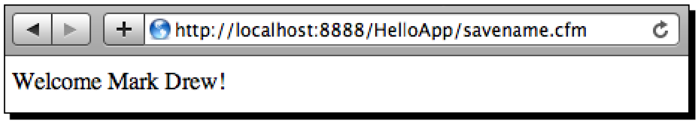
To define an **Application**, we need to create a special template called `Application.cfc` in the root of our `HelloApp` folder. This template manages our application lifecycle and various other settings that we can use within our application. Let's define it now:

1. In the `HelloApp` folder, create a file called `Application.cfc` with the following code:

```
<cfcomponent output="false">
  <cfset this.name = "RememberName">
  <cfset this.sessionmanagement = true>
</cfcomponent>
```

In this template, we have simply defined a component with the `<cfcomponent>` tag. Inside this tag, we set two variables: `this.name="RememberMe"`, which will be the name of our application, separating it out from other applications on the server, and `this.sessionmanagement = true`, which sets whether this application will manage sessions or not. By setting it to `true`, we now have access to the `Session` scope.

2. In the form, re-enter your name and submit the form. You will now get a friendly greeting.



Welcome Mark Drew!

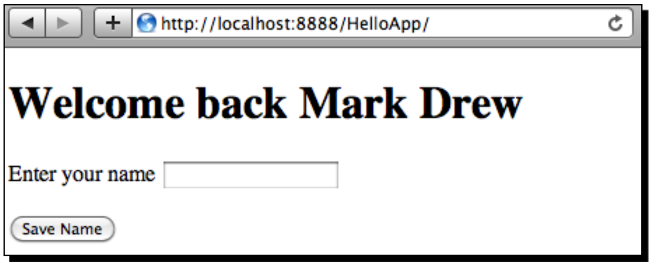
But how do we know that we have actually set this name for the whole session?

3. Change the `index.cfm` template to display a `Welcome back` message, if we have set the name:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Remember My Name</title>
  </head>
  <body>
    <cfparam name="SESSION.name" default="">
    <cfif Len(SESSION.name)>
      <h1>Welcome back <cfoutput>#SESSION.name#</cfoutput></h1>
    </cfif>
    <form action="savename.cfm" method="post">
      <label>Enter your name</label>
      <input type="text" name="name" value="">
      <p><input type="submit" value="Save Name"></p>
    </form>
  </body>
</html>
```

In this code, we set a default parameter for the `SESSION.name` variable. Then, we check to see if there is any length to the string value and then display it.

4. Go back to the `Helloapp` application, without submitting your name. You will see that the application has remembered your name:



Welcome back Mark Drew

Enter your name

What just happened?

The Application.cfc component manages various settings of your application. By adding it to the HelloApp folder, we said that all the templates (and templates in directories below the current directory) are part of our application. Any setting, such as the this.sessionmanagement setting, will now apply to these templates.

Session and client settings

You can control other session settings in your Application.cfc too. There are a number of session and client information storage settings; a few of them are listed as follows:

Setting	Description
applicationTimeout	Defines how long variables in the APPLICATION scope will persist
sessionManagement	Whether there will be a SESSION scope available to the application
sessionTimeout	Defines how long variables in the SESSION scope will persist
setClientCookies	Whether the application will set client cookies
setDomainCookies	Should the cookies be domain-specific
clientManagement	Whether we should use the CLIENT scope, such as the COOKIES scope stored on the server
clientStorage	Where the CLIENT scope variables are stored

Have a go hero

Since we have told you what all the settings are, why don't you try them out? There are clues in the names, of course, and if you are stuck, check out the Railo Web Administrator sections that we went over in the previous chapter.

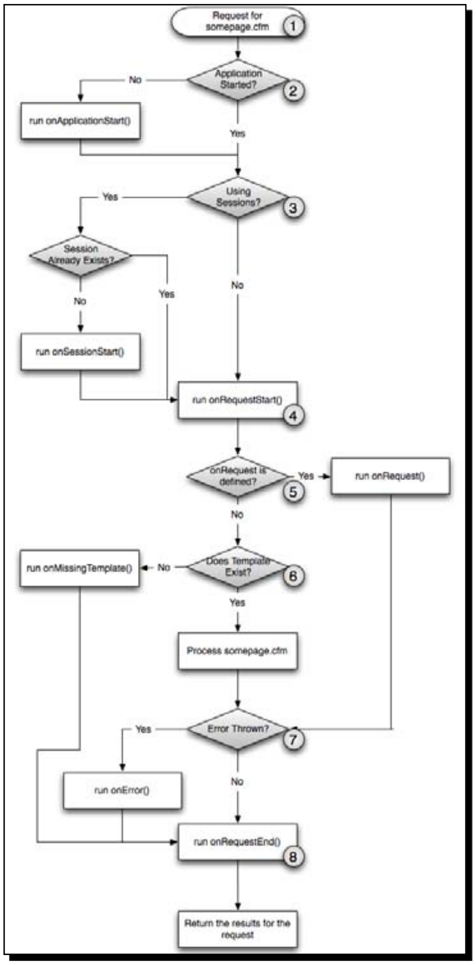
Application events

Apart from being able to manage settings of your application in the Application.cfc, you can also manage different parts of the Application, Session, Request, and Error lifecycle. To do this, we can create functions within the Application.cfc that are triggered at different points.

There are a number of methods that you can implement in `Application.cfc`. They are simply functions. For example, the `onApplicationStart()` method will be executed if the application hasn't started up:

```
<cfcomponent>
  <cfset this.name = "RememberName">
  <cfset this.sessionmanagement = true>
  <cffunction name="onApplicationStart" returnType="boolean"
    output="false">
    <cfreturn true>
  </cffunction>
</cfcomponent>
```

Before trying out some of these functions, let's look at the Application lifecycle and how Railo Server triggers different functions. The following screenshot depicts the Application lifecycle:



1. When an initial request arrives at the web server, it checks the extension. If it is a `.cfm` or a `.cfc`, it gets passed onto Railo Server.
2. When Railo Server receives a request, it checks if there is an `Application.cfc` file. If there is one, it will check if there is an application defined, and set all the variables and settings. If there is an `onApplicationStart()` method, it will run the method, for example:

   ```
   <cfcomponent>
     <cfset this.name = "RememberName">
     <cfset this.sessionmanagement = true>
     <cffunction name="onApplicationStart" returnType="boolean"
       output="false">
       <cfset application.mySetting = "MySetting">
       <cfreturn true>
     </cffunction>
   </cfcomponent>
   ```

   In the `onApplicationStart()` method, you can set any of the `APPLICATION` scope variables that your application might need. This will run only if the application hasn't started or if there have been no requests to the server for longer than the `this.applicationtimeout` setting.

   When the application timeout time has elapsed, you can also define an `onApplicationEnd` function that will be called. Remember that at this point, the code will NOT have access to the `APPLICATION` scope, but you can get the variables from the, now elapsing, scope, as they are passed into the function through the `ARGUMENTS.applicationScope` structure:

   ```
   <cffunction name="onApplicationEnd" returnType="void"
     output="false">
     <cfargument name="applicationScope" required="true">
     <!--- do something --->
   </cffunction>
   ```

   This can be useful to, for example, log variables that you have kept in the `APPLICATION` scope or to do any other clean up function.
3. Railo Server then checks to see if `sessionmanagement` has been turned on in this `Application`. If it is turned on and the current user doesn't have a session yet, then it will run the `onSessionStart()` method, where you can set any variables that the user may need for this session, such as the defaults for example:

   ```
   <cffunction name="onSessionStart" returnType="void"
     output="false">
     <cfset SESSION.setting = "User Session Setting">
   </cffunction>
   ```

4. Analogous to the `onApplicationEnd`, there is also an `onSessionEnd` that will be run when a user's session has elapsed, and again, you won't be able to access the `SESSION` scope directly, but you will have access to the `ARGUMENTS.sessionScope` variable:

```
<cffunction name="onSessionEnd" returnType="void" output="false">
  <cfargument name="sessionScope" type="struct" required="true">
  <cfargument name="appScope" type="struct" required="false">
</cffunction>
```

5. Now that the Application and Session-related start functions have been called, you can also implement an `onRequestStart` method to add any variables or settings related to that request:

```
<cffunction name="onRequestStart" returnType="boolean"
  output="false">
  <cfargument name="thePage" type="string" required="true">
  <cfreturn true>
</cffunction>
```

6. You can also implement an `onRequest` method, but be warned, this means that "your" code will manage including a page, rather than be part of the normal flow of a request. This can be rather confusing if you have no code in there, since nothing will be returned.

7. Railo Server now checks to see if the requested template exists in the file system. If the template is not found, the error page defined in the Railo Web Administrator will be displayed. But if you implement an `onMissingTemplate`, then you can implement your own code specifically for this application, for example, you could log the missing template or return a different one, as shown below:

```
<cffunction name="onMissingTemplate" returnType="boolean"
  output="true">
  <cfargument name="targetpage" required="true" type="string">
  <cfinclude template="404.cfm">
  <cfreturn true>
</cffunction>
```

8. Now Railo Server processes the page that was requested. If there is an error thrown, Railo will run the template defined in the Railo Web Administrator, but you can override this behavior by implementing an `onError` method:

```
<cffunction name="onError" returnType="void" output="true">
  <cfargument name="exception" required="true">
  <cfargument name="eventname" type="string" required="true">
  <cfinclude template="MyError.cfm">
</cffunction>
```

9. Once Railo has processed any errors, you can implement the `onRequestEnd` method. This will do any post-processing (maybe log the type of request the user requested):

    ```
    <cffunction name="onRequestEnd" returnType="void" output="false">
      <cfargument name="thePage" type="string" required="true">
      <!--- do something --->
    </cffunction>
    ```

    And finally, the rendered template is now passed back to the user.

We have covered some of the functionality of the `Application.cfc` file. You should now have a good idea of how a request is processed by the `Application.cfc` file.

Pop quiz – Application.cfc

Time to get the old brain cells thinking!

Can you put these functions in the order that they are called?

```
onRequest

onRequestStart

onSessionStart

onError

onApplicationEnd

onApplicationStart
```

Object relational mapping with Railo Server

Applications can get complex as they get larger. It is fine to just do a few templates with inline queries, but as your applications increase in size, developers realize that keeping business logic in the correct place becomes harder.

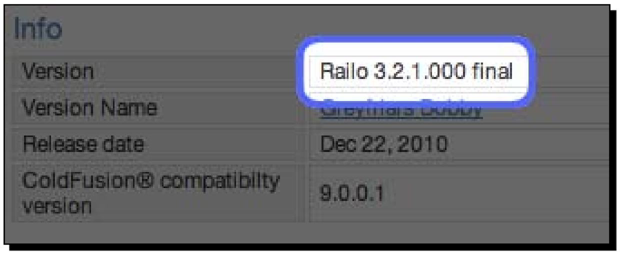
Instead of talking of displaying our tables in a web page, why not talk about discreet objects and how they are related to each other? Once we have done that, we will want the objects to somehow persist for longer than the request that created them, so why not store them in a database? This is incredibly simple in Railo Server, as we shall see.

Railo Server allows you to do this with the use of the **Hibernate Persistence** from **JBoss** (`http://www.hibernate.org/`), but without you having to do a lot of the configuration that is normally required to use it in Java applications.

Time for action – upgrading Railo Server

Before we get started, it should be mentioned that the ORM feature is in the **Beta** state for Railo 3.2, but this is ok. Let's update the Railo Server, so that we can use these features. It's quick and painless:

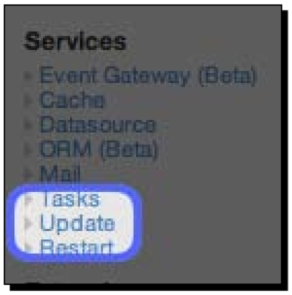
1. Open the Railo Administrator at `http://localhost:8888/railo-context/admin/index.cfm` and click on the **Server Administrator** tab.
2. Either enter your password or set a password if you have not already done so.
3. Notice the current version of Railo Server in your installation:



Info

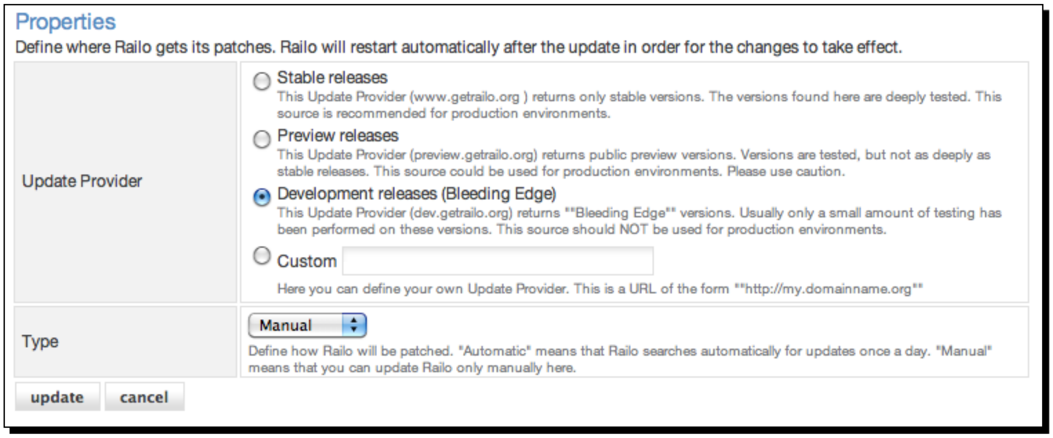
Version	Railo 3.2.1.000 final
Version Name	Greyfriars Bobby
Release date	Dec 22, 2010
ColdFusion® compatibilty version	9.0.0.1

4. Under the **Services** section, click on the **Update** link.



5. The **Update** section allows you to stay up-to-date with major and minor releases of Railo Server.

   In the **Properties** section, select the **Development releases** (**Bleeding Edge**) radio button and click on the **Update** button.



Properties

Define where Railo gets its patches. Railo will restart automatically after the update in order for the changes to take effect.

Update Provider

○ Stable releases
This Update Provider (www.getrailo.org) returns only stable versions. The versions found here are deeply tested. This source is recommended for production environments.

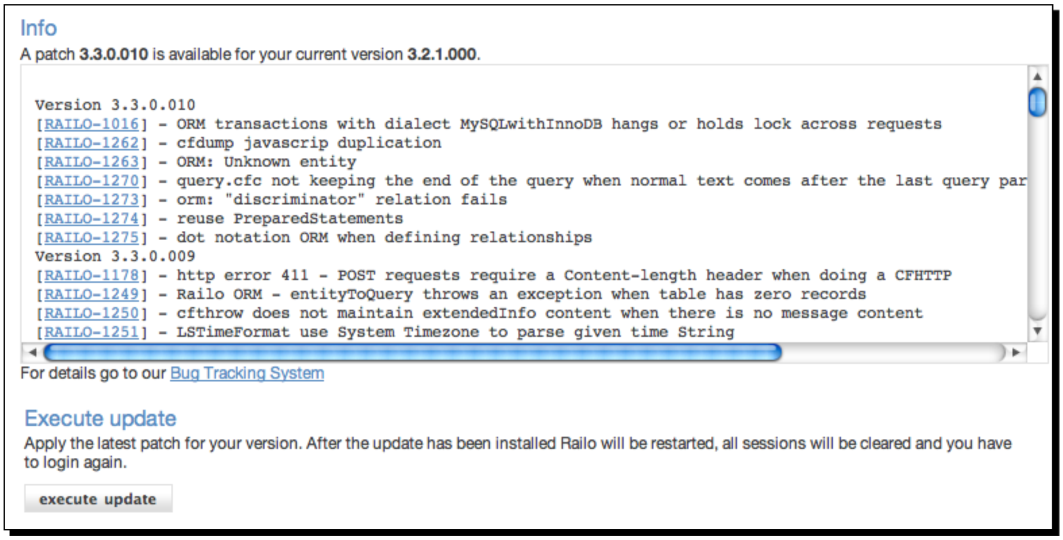
○ Preview releases
This Update Provider (preview.getrailo.org) returns public preview versions. Versions are tested, but not as deeply as stable releases. This source could be used for production environments. Please use caution.

● Development releases (Bleeding Edge)
This Update Provider (dev.getrailo.org) returns ""Bleeding Edge"" versions. Usually only a small amount of testing has been performed on these versions. This source should NOT be used for production environments.

○ Custom
Here you can define your own Update Provider. This is a URL of the form ""http://my.domainname.org""

Type: Manual
Define how Railo will be patched. "Automatic" means that Railo searches automatically for updates once a day. "Manual" means that you can update Railo only manually here.

6. Once the page refreshes, scroll down and you should be able to see an **Info** panel, showing you the release notes of the latest version of Railo. Click on the **execute update** button; this will install the latest version of Railo Server.



Info

A patch **3.3.0.010** is available for your current version **3.2.1.000**.

```
Version 3.3.0.010
[RAILO-1016] - ORM transactions with dialect MySQLwithInnoDB hangs or holds lock across requests
[RAILO-1262] - cfdump javascrip duplication
[RAILO-1263] - ORM: Unknown entity
[RAILO-1270] - query.cfc not keeping the end of the query when normal text comes after the last query par
[RAILO-1273] - orm: "discriminator" relation fails
[RAILO-1274] - reuse PreparedStatements
[RAILO-1275] - dot notation ORM when defining relationships
Version 3.3.0.009
[RAILO-1178] - http error 411 - POST requests require a Content-length header when doing a CFHTTP
[RAILO-1249] - Railo ORM - entityToQuery throws an exception when table has zero records
[RAILO-1250] - cfthrow does not maintain extendedInfo content when there is no message content
[RAILO-1251] - LSTimeFormat use System Timezone to parse given time String
```

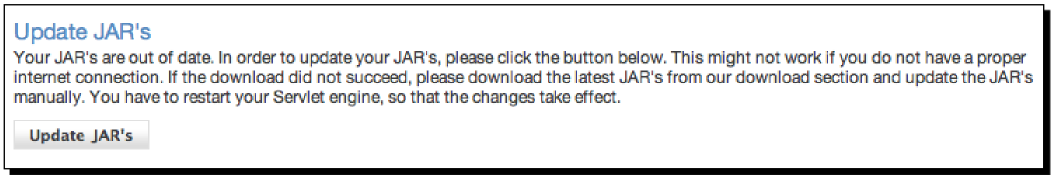
For details go to our Bug Tracking System

Execute update

Apply the latest patch for your version. After the update has been installed Railo will be restarted, all sessions will be cleared and you have to login again.

7. Once the server has updated, you will have to log in again, and you should now be on the latest version of Railo Server. Congratulations!

8. If you see a section about updating your JARs, just click the **Update JAR's** button (if you are running Windows, you might have to restart the Railo Server by shutting it down in the console and clicking on the **start.bat** or re-running the start script under your Railo Server install directory):



Update JAR's

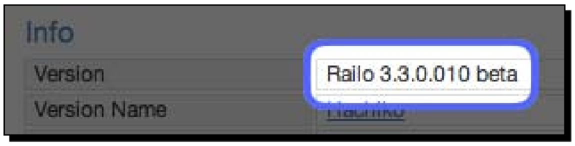
Your JAR's are out of date. In order to update your JAR's, please click the button below. This might not work if you do not have a proper internet connection. If the download did not succeed, please download the latest JAR's from our download section and update the JAR's manually. You have to restart your Servlet engine, so that the changes take effect.

Wow! That was pretty easy!

What just happened?

By going to the Server Administrator, we were able to update Railo Server to the latest version and get new features; we didn't have to install anything or go and download anything ourselves, the Railo Server Administrator did it for us.

If you now check the Railo Server Administrator homepage by going to `http://localhost:8888/railo-context/admin/server.cfm`, you should see that we have now got the latest (and greatest!) version of Railo Server. We can now start playing with the awesome ORM capabilities.



Info

Version Railo 3.3.0.010 beta

Version Name

Creating our database persistence store

To demonstrate the features of the ORM, we are going to quickly build a simple blog. The blog will have posts, and people will be able to comment on each post. This is just going to be a simple example to show you some of the capabilities of the ORM, rather than trying to implement a production-ready blog.

Let's get started!

Time for action – creating a database

Before we can start persisting our components into a database, we have to actually tell Railo Server about that database. In your MySQL database, create a new empty database, either by using a **Graphical User Interface** (**GUI**) tool or by using the command line utilities:

1. At the command line, enter the following command to log into the `mysql` console:

   ```
   > mysql -u root -p
   ```

2. When prompted, enter your password.
3. Enter the following to create a database called `railoblog`:

   ```
   > CREATE DATABASE railoblog;
   ```

4. `mysql` should reply with:

   ```
   Query OK, 1 row affected (0.10 sec)
   ```

What just happened?

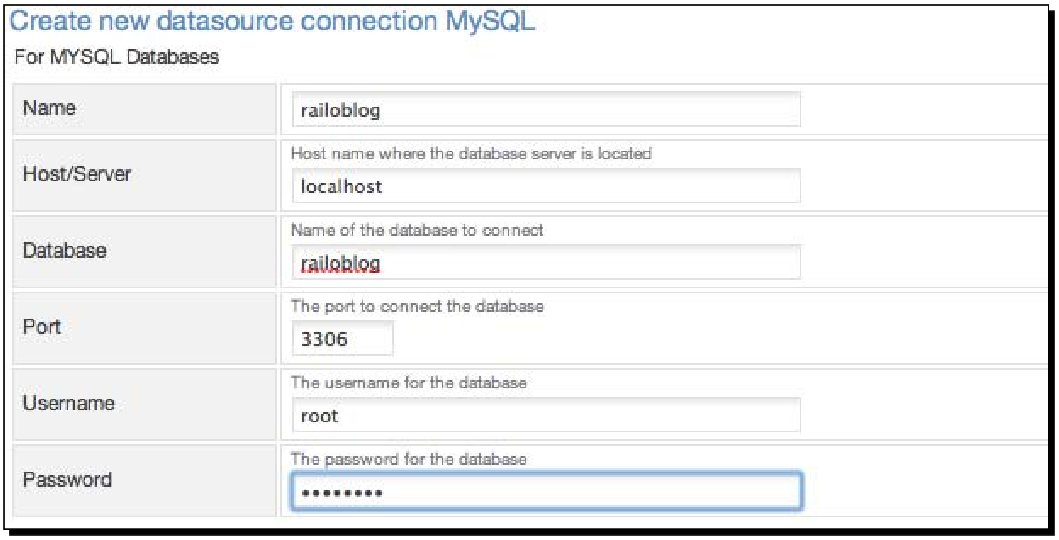
We have just created a database that we are going to connect to from our templates. Let's go and configure the datasource now.

Time for action – creating our railoblog datasource

So that we can connect to our database from our application, we need to set up a datasource in the Railo Web Administrator. Let's set that up now, by carrying out the following steps:

1. In your browser, open the Railo Web Administrator by going to `http://localhost:8888/railo-context/admin/web.cfm`.
2. Click **Datasource** under the **Services** section.
3. Enter the name `railoblog` in the **Name** field of the **Create new datasource** form, select **MySQL** from the **Type** drop-down, and click on **Create**.

4. Since we are doing local development, the **Host/Server** and **Port** in the **Create new datasource connection MySQL** form should be fine. Enter the **Database** name as `railoblog` and enter values for the **Username** and **Password** to connect to the server:



Create new datasource connection MySQL

For MYSQL Databases

Name	railoblog
Host/Server	Host name where the database server is located localhost
Database	Name of the database to connect railoblog
Port	The port to connect the database 3306
Username	The username for the database root
Password	The password for the database •••••••••

5. Finally, click on the **Create** button at the end of the form.

What just happened?

We have created a datasource that points to our newly-created database. We can now use this datasource to create our application

Using persistent components

To create our blog, we are going to need to set up our `Application.cfc` and define some components that will be persisted to the database. Let's do that now.

Time for action – creating the blog

Let's get down to creating our blog. To keep it simple, we are only going to have one main page that will show our posts. We will get a few blog posts and comments and display those.

1. In your `<Railo Install Directory>/webroot` directory, create a new folder called `blog`.

2. Create the main `index.cfm` page with the following code:

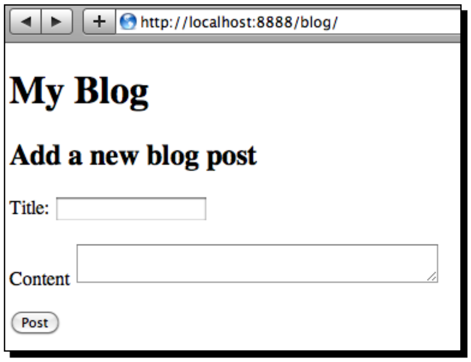
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Blog</title>
    <meta name="author" content="Mark Drew">
  </head>
  <body>
    <h1>My Blog</h1>
    <h2>Add a new blog post</h2>
    <form action="addpost.cfm" method="post">
      <p>
        <label for="title">Title:</label>
        <input type="text" name="title" value="" id="title">
      </p>
      <p>
      <label for="content">Content</label>
        <textarea name="content" rows="content" cols="40"
          id="content">
        </textarea>
      </p>
      <p>
        <input type="submit" value="Post">
      </p>
    </form>
  </body>
</html>
```

This code is just our form for adding our new posts.

3. Now let's create the page where we will be saving our blog posts, and name it `addpost.cfm` (as defined in form action):

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Post Saved</title>
    <meta name="author" content="Mark Drew">
  </head>
  <body>
    <h1 id="post_saved!">Post Saved!</h1>
    <cfdump var"#FORM#" />
    <p>
      <a href="index.cfm">Back</a>
    </p>
  </body>
</html>
```

We should now have a nice form that we can submit, which looks as follows:



My Blog

Add a new blog post

Title:

Content

4. So far, these are just two templates. Let's make this an application and configure the ORM capabilities. Let's create an `Application.cfc` in this folder too, with the following content and settings:

```
<cfcomponent output="false">
  <cfset this.name = "MyBlog">
  <cfset this.datasource = "railoblog">
  <cfset this.ormEnabled= true>
  <cfset this.ormSettings.dbcreate = "dropcreate">
</cfcomponent>
```

We set the name of the application to `MyBlog`, then we set the datasource, enabled the ORM with `this.ormEnabled =true`, and then set the `ormSettings` to `dropcreate`, which will delete the table and create it when changes are made. This is ok for the moment, since we are developing and we don't care about the data we are storing.

5. Great! Now that the ORM is configured, let's create a persistent **Post** object that we are going to save in our `addpost.cfm` page. Create a file called `post.cfc`:

```
<cfcomponent persistent="true" entityname="post" output="false">
  <cfproperty name="id" ormtype="id" generator="native">
  <cfproperty name="title" ormtype="string">
  <cfproperty name="content" ormtype="text">
  <cfproperty name="dateCreated" fieldtype="timestamp">
</cfcomponent>
```

In this code, you can see the component tag with a new attribute of `persistent="true"`, which tells Railo Server that we are going to persist it to the ORM. We give the entity the name of the post in the `entityname="post"` attribute. Now to define the properties of our object, we add some properties using the `<cfproperty>` tag, and they all are given a name. The `id` property is given the `ormtype="id"` to define it as a unique identifier.

The `title` property is given the `ormtype="string"` to say it will be stored as a `varchar`.

The `content` property is given the `ormtype="text"` to say that it will be a long bit of text.

Finally, we create a `dateCreated` property, and say it's a special type of field called `timestamp` that will put in the current time and date when it is saved.

That's it! We have defined our persistent object. Let's go and save our form to our database now.

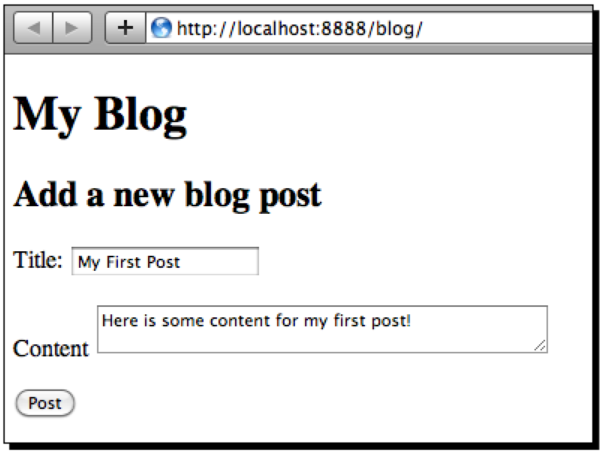
6. Let's modify our `addpost.cfm` template, so that it saves the form fields to the ORM:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Post Saved</title>
    <meta name="author" content="Mark Drew">
  </head>
  <body>
 <cfset myPost = EntityNew("post")>
 <cfset myPost.setTitle(FORM.title)>
 <cfset myPost.setContent(FORM.content)>
 <cfset EntitySave(myPost)>
 <h1 id="post_saved!">Post Saved!</h1>
    <p><a href="index.cfm">Back</a></p>
  </body>
</html>
```

In the first line, we created a new post using `EntityNew("post")`, which is a blank persistent entity. Automatically, getters and setters will be defined for our object. So if you want to set the value of the title property, all you have to do is call `myPost.setTitle("..")`, and if you want to get a value from an existing object, all you have to do is call `myPost.getTitle()`.

Once we have set the `title` and `content` to our object, all we have to do is save the `myPost` object (we do that with `EntitySave(myPost)`).

7. Now that we have done that, we can go back to our form, fill it with some content, and submit it:



http://localhost:8888/blog/

My Blog

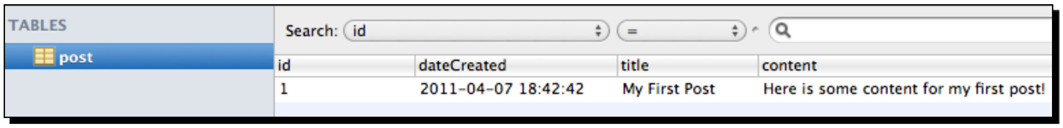
Add a new blog post

Title: My First Post

Content: Here is some content for my first post!

8. Once we have submitted it, you will get a page displaying the message **Post Saved!**.

9. If you now go and check in your database with a GUI tool, you will see that the object has been persisted.



TABLES

post

Search: id =

id	dateCreated	title	content
1	2011-04-07 18:42:42	My First Post	Here is some content for my first post!

What just happened?

Wow, in just a few lines of code, we have used **Hibernate** to persist our objects. What we managed to do in just a few lines of code would have been much harder to set up with Java, but Railo Server made it easy for us.

As we saw before, we added an `ormEnable` setting in our `Application.cfc` and we set the datasource and the settings that the ORM should use when our objects change. Then we defined our persistent object, just not like any component using the `cfproperty` tag, but by adding some information that enabled the ORM to create the tables for us. And finally, we created a new object, filled it with data, and saved it to the database.

Time for action – listing our blog posts

It's all fine to be creating posts in your blog, but if no one can see them, there is no point. Let's customize it, so that everyone can see our posts:

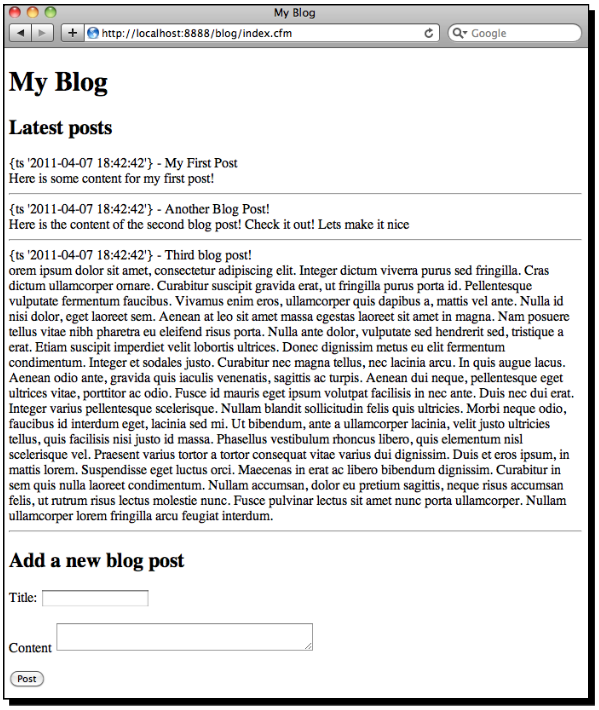
1. Let's add some code to our `index.cfm` to list our blog posts:

```
<!DOCTYPE html>
…
  <body>
    <h1>My Blog</h1>
 <h2>Latest posts</h2>
 <cfset Posts = EntityLoad("post")>
   <cfloop array="#Posts#" index="post">
   <cfoutput>
     <div class="post">
       <div class="title">#post.getDateCreated()# -
         #post.getTitle()#</div>
       <div class="content">#post.getContent()#</div>
     </div>
     <hr>
   </cfoutput>
 </cfloop>
 <h2>Add a new blog post</h2>
…
```

In this code, we have got our `Posts` using the `EntityLoad("post")`. This returns an array of `Post` objects. We then use `<cfloop>` to go through the posts and use `<cfoutput>` around our code to output the variables from the object.

Have you noticed the `index="post"` in the `<cfloop>` tag? Well this is going to be the variable that is going to hold each blog post that we can then use to get the properties out of using the getters; so for example, we get the `post.getContent()` to get the value stored in content.

2. After adding a few more posts into the blog, you should be able to see all the items being output:



My Blog

Latest posts

{ts '2011-04-07 18:42:42'} - My First Post
Here is some content for my first post!

{ts '2011-04-07 18:42:42'} - Another Blog Post!
Here is the content of the second blog post! Check it out! Lets make it nice

{ts '2011-04-07 18:42:42'} - Third blog post!
orem ipsum dolor sit amet, consectetur adipiscing elit. Integer dictum viverra purus sed fringilla. Cras dictum ullamcorper ornare. Curabitur suscipit gravida erat, ut fringilla purus porta id. Pellentesque vulputate fermentum faucibus. Vivamus enim eros, ullamcorper quis dapibus a, mattis vel ante. Nulla id nisi dolor, eget laoreet sem. Aenean at leo sit amet massa egestas laoreet sit amet in magna. Nam posuere tellus vitae nibh pharetra eu eleifend risus porta. Nulla ante dolor, vulputate sed hendrerit sed, tristique a erat. Etiam suscipit imperdiet velit lobortis ultrices. Donec dignissim metus eu elit fermentum condimentum. Integer et sodales justo. Curabitur nec magna tellus, nec lacinia arcu. In quis augue lacus. Aenean odio ante, gravida quis iaculis venenatis, sagittis ac turpis. Aenean dui neque, pellentesque eget ultrices vitae, porttitor ac odio. Fusce id mauris eget ipsum volutpat facilisis in nec ante. Duis nec dui erat. Integer varius pellentesque scelerisque. Nullam blandit sollicitudin felis quis ultricies. Morbi neque odio, faucibus id interdum eget, lacinia sed mi. Ut bibendum, ante a ullamcorper lacinia, velit justo ultricies tellus, quis facilisis nisi justo id massa. Phasellus vestibulum rhoncus libero, quis elementum nisl scelerisque vel. Praesent varius tortor a tortor consequat vitae varius dui dignissim. Duis et eros ipsum, in mattis lorem. Suspendisse eget luctus orci. Maecenas in erat ac libero bibendum dignissim. Curabitur in sem quis nulla laoreet condimentum. Nullam accumsan, dolor eu pretium sagittis, neque risus accumsan felis, ut rutrum risus lectus molestie nunc. Fusce pulvinar lectus sit amet nunc porta ullamcorper. Nullam ullamcorper lorem fringilla arcu feugiat interdum.

Add a new blog post

Title:

Content

3. Wow! That was easy to get a list of `Posts`. But wait, there is a problem here. The order of the blog posts is wrong. We want to order them by the `dateCreated`, so that the latest post is at the top. Let's change the `EntityLoad("post")` to the following:

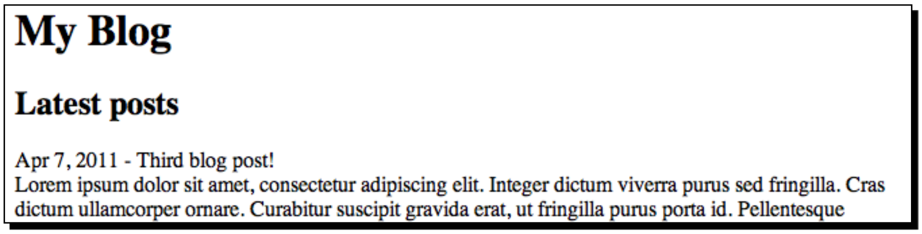
```
<cfset Posts = EntityLoad("post", {}, "dateCreated desc")>
```

The function `EntityLoad` now takes three parameters, namely, the name of the entity, a filter that will only return items that match key-value pairs, and finally the ordering commands (in this case, we want the posts sorted by date). This now returns our `Posts` with the latest at the top.

4. Let's clean up the date. Since there is an automatically created function for `getDateCreated`, we can actually override this. Let's do that to return a nicer looking date. In your `post.cfc`, add the following function:

```
<cffunction name="getDateCreated">
  <cfreturn DateFormat(dateCreated,"medium")>
</cffunction>
```

The dates will now be output using the `DateFormat()` function:



My Blog

Latest posts

Apr 7, 2011 - Third blog post!
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer dictum viverra purus sed fringilla. Cras dictum ullamcorper ornare. Curabitur suscipit gravida erat, ut fringilla purus porta id. Pellentesque

What just happened?

In the previous section, we managed to list all our posts using the `EntityLoad()` function, which brought back all of our posts from the database. Since they weren't in the right order (with the latest at the top), we added a third parameter to our `EntityLoad("Post", {}, "dateCreated desc")` function to specify which property we want to order by. Finally, since we are creating a timestamp for our `dateCreated` property, we overrode the built-in getter and formatted the date nicely.

Time for action – adding comments

A blog wouldn't really be a blog if users can't make comments to a post. We are going to now add comments to our posts by relating a post to a comment object.

1. First thing to do is to create a comment persistent object. Let's create a template called `Comment.cfc` with the following description:

```
<cfcomponent persistent="true" entityname="comment"
output="false">
  <cfproperty name="id" ormtype="id" generator="native">
  <cfproperty name="from" ormtype="string">
  <cfproperty name="comment" ormtype="text">
  <cfproperty name="dateCreated" fieldtype="timestamp">
  <cfproperty name="post" fieldtype="one-to-one" fkcolumn="id"
    cfc="Post" insert="false" update="false">
</cfcomponent>
```

We have seen most of the properties before, except that now a `Comment` has a property called `post` that is related to the `Post` object.

- First, we add a `fieldtype="one-to-one"` to the property to define how this component will relate in the ORM to the `Post` object.
- Now, let's define which column will be referenced in the foreign key column in the `Post` object using `fkcolumn="id"`.

- Then we say which is the component we are relating to with the `cfc="Post"` attribute.
- Finally, we set the `insert="false"` and `update="false"` attributes. These define whether if we save a `Comment` they will update the `Post` object, which we don't need to do.

2. Now that we have related our `Comment` to our `Post` object, let's add the inverse relationship from the `Post` to the `Comment` objects. After all, we want to get all the comments for a `Post`. Edit the `Post.cfc` and add the following property:

```
<cfproperty name="comments" type="array" fieldtype="one-to-many"
  cfc="Comment" fkcolumn="postid">
```

3. We have created a property called `comments` and defined its type as an array of comment components with `cfc="Comment"`. We defined the relationship is a one-to-many relationship with `fieldtype="one-to-many"`. Finally, we link it to the `postid` of the `Comment` column using `fkcolumn="postid"`.

4. Time for us to add a form so that users can post a comment. Let's add a comment form under each `Post`:

```
<div class="newComment">
  <form action="addComment.cfm" method="post">
    <input type="hidden" name="postid" value="#post.getId()#"
      id="postid">
    <p>
      <label for="name">Name</label>
      <input type="text" name="name" value="" id="name">
    </p>
    <p>
      <label for="comment">Comment</label>
      <textarea name="comment" rows="comment" cols="40"
        id="comment">
      </textarea>
    <p>
    <p><input type="submit" value="AddComment"></p>
  </form>
</div>
```

This form is pretty standard; it points to the `addComment.cfm` template (this will save our comment) and it has a hidden field called `postid` that contains the `id` of the current `Post`.

5. Let's create the `addComment.cfm` template. This will save our comments:

```
<cfparam name="FORM.postid" type="numeric">
<cfset post = EntityLoad("post", FORM.postid, true)>
<cfset comment = EntityNew("comment")>
<cfset comment.setFrom(FORM.name)>
<cfset comment.setComment(FORM.comment)>
<cfset EntitySave(comment)>
<cfset post.addComments(comment)>
<cfset EntitySave(post)>
<cflocation url="index.cfm" addtoken="false">
```

In this code, we:

- Make sure that a `postid` is passed in the `FORM` scope with the `<cfparam>` tag and that it is a numeric value (an error will be thrown if it isn't).
- Then, we load up the `Post` entity with the `id` of `FORM.postid` using the `EntityLoad()` function. The last attribute passed to `EntityLoad()` (`true`) says that we only want one `Post` returned.
- Then we create a new `Comment` entity and fill in the variables, using the `setFrom()` and `setComment()` methods, and then save it using the `EntitySave()` function.
- We then add the comment to the post using the `addComments()` method that has been automatically generated for us. Finally, we save the post object, again using the `EntitySave()` function.
- Since we don't need to show any output, we then relocate to the `index.cfm` using the `<cflocation>` tag.

6. Now that we have added a comment, the final task is to actually list each of the comments for a post. After the post, we can get the comments using the `autogenerated` method of `post.getComments()`.

Let's add some code to display the comments after each post in `index.cfm`:

```
<div class="comments">
  <cfloop array="#post.getComments()#" index="comment">
    <div class="commment">
      <div class="commentFrom">#comment.getFrom()# on
        #comment.getDateCreated()#
      </div>
      <div class="commentText">#comment.getComment()#</div>
    </div>
  </cfloop>
</div>
```

In this code, we get the comments and loop through them, passing them to a comment variable using `index="comment"` in the `<cfloop>` tag. For each comment, we simply output the `From`, `DateCreated`, and `Comment` properties using the auto-generated getters.

What just happened?

In the previous section, we saw how we can relate objects to each other quite simply by adding a `fieldtype` to a component's property, including a back relationship. The relationships that you can have between objects are:

- one-to-one
- one-to-many
- many-to-one
- many-to-many

You can then use various related objects with methods, such as `getComments`, `addComment`, and `hasComments` to find out what kind of related objects are assigned to a primary object with the relations you have set.

This is just a taste of the ORM capabilities in Railo Server.

Caching in Railo Server

Getting and displaying content from a database is pretty easy in Railo Server. You can use the ORM capabilities or use queries with the `<cfquery>` tag, and you can develop applications relatively quickly.

Unfortunately, the world has other plans. Once you put your application live, you will notice that different parts of your application can start to become bottlenecks in the overall response time. When you start analyzing what is slowing things down, you will soon discover that sometimes your application is busy doing things that is has already done before. For example, returning a list of countries for every request. This data has not changed (unless there is a big change in the geo-political landscape of the world) and probably won't for a few years.

Therefore, it makes a lot of sense to cache this content once and serve it from the cache for the subsequent requests. Of course, caching has some limitations, since some elements cannot be cached or need to be most recent, but for now, we will just focus on elements that can be cached.

Railo Server allows you to create different caches that allow you to store data; but before we look into the details, let's get a feel for what a cache is good for.

Cache: what is it good for?

The Railo cache allows data that has been written to a database, a variable, or to the filesystem to now be stored and retrieved without having to get it again from the original source. The main advantages of this are:

- Access is much faster than reading and writing from the filesystem
- You can determine the lifetime of the elements that are stored in the cache
- The data can be (but does not need to be) persistent, that is, it survives a restart of Railo Server
- The memory used for storing items is quite small in comparison to data stored in a variable, since caches usually have an intelligent paging mechanism
- Data can be easily distributed across multiple systems (peer-2-peer) or can be centrally maintained (client-server), so that multiple Railo Server instances can have access to the same objects

Time for action – creating a cache connection

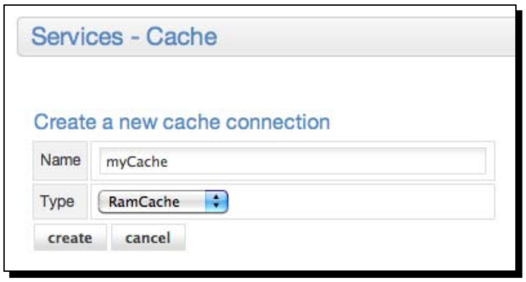
Before we can see the power of using caching services, we need to define a cache connection.

In the Railo (Server and Web) Administrators, you can create and manage cache instances. Railo Server allows you to create as many instances as you need. The concept is similar to the creation of a datasource connection under Services/Datasource.

Let's go and create a cache connection:

1. Head to the Railo Web Administrator by going to `http://localhost:8888/railo-context/admin/web.cfm`

2. Log in if you have not already done so, and then click on the **Cache** link under **Services**. You will see something similar to the following screenshot:



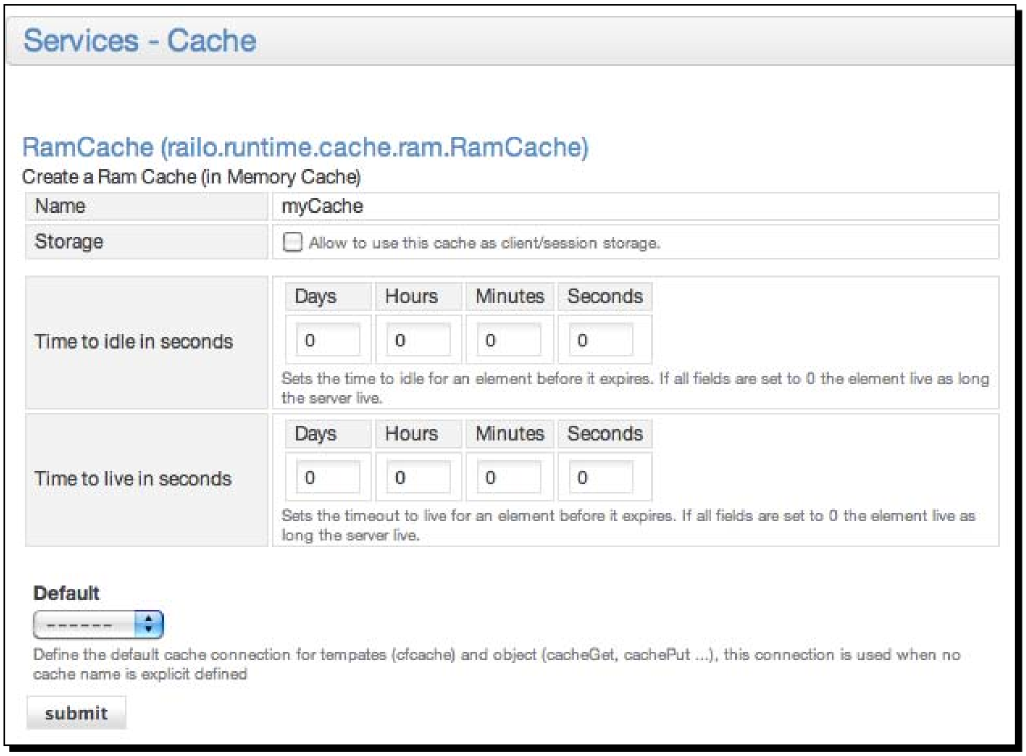
Services - Cache

Create a new cache connection

Name	myCache
Type	RamCache

Out of the box, Railo Server provides the options to create a **RamCache** or an **EHCache Light Cache Connections**. **RamCache** is the default cache implementation for Railo Server. Let's create a **Cache** connection by giving it a name (without spaces and special characters). Let's call it **myCache** and select the **RamCache** connection. Click on the **create** button.

3. Afterwards, we see the details of the cache. We are presented with the configuration details of our cache, as shown in the following screenshot:



Services - Cache

RamCache (railo.runtime.cache.ram.RamCache)

Create a Ram Cache (in Memory Cache)

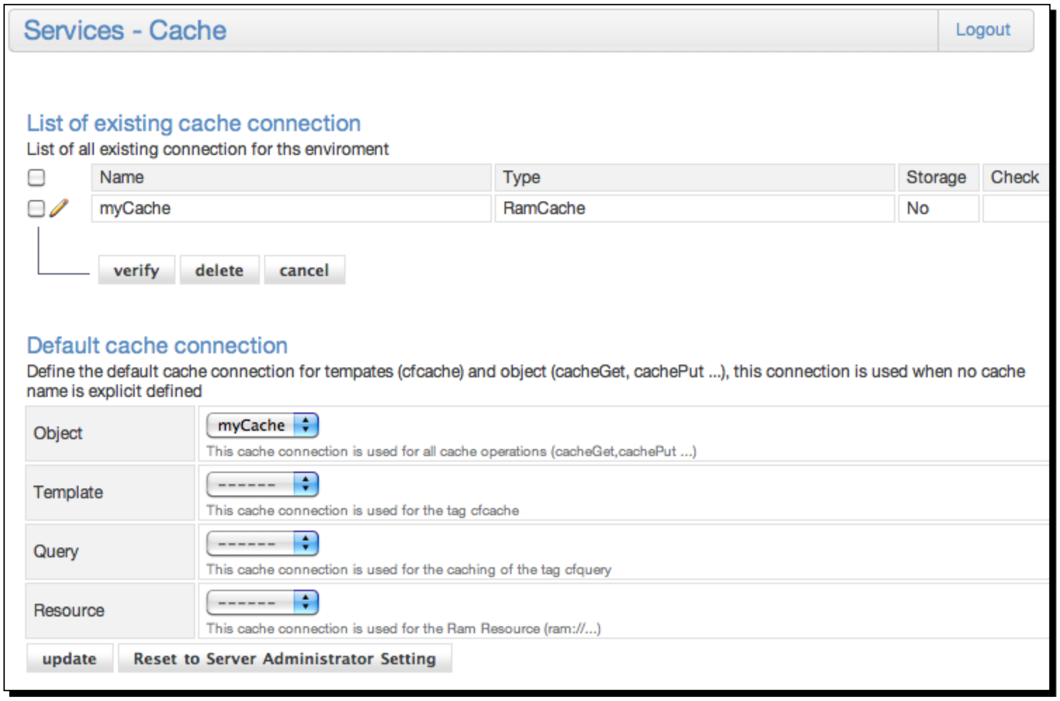
Name	myCache
Storage	□ Allow to use this cache as client/session storage.
Time to idle in seconds	Days 0, Hours 0, Minutes 0, Seconds 0 Sets the time to idle for an element before it expires. If all fields are set to 0 the element live as long the server live.
Time to live in seconds	Days 0, Hours 0, Minutes 0, Seconds 0 Sets the timeout to live for an element before it expires. If all fields are set to 0 the element live as long the server live.

Default

Define the default cache connection for tempates (cfcache) and object (cacheGet, cachePut ...), this connection is used when no cache name is explicit defined

The **RamCache** only has two settings, namely, **Time to idle in seconds** and **Time to live in seconds**. The final setting states what type of cache connection this should be the default for. Since we are going to be using it for objects, let's select **object** and click on **submit**.

4. We are now presented with a list of our defined cache connections. See the new cache is listed in the following screenshot:



List of existing cache connection

List of all existing connection for ths enviroment

□	Name	Type	Storage	Check
□	myCache	RamCache	No	

Default cache connection

Define the default cache connection for tempates (cfcache) and object (cacheGet, cachePut ...), this connection is used when no cache name is explicit defined

Object	myCache This cache connection is used for all cache operations (cacheGet,cachePut ...)
Template	------ This cache connection is used for the tag cfcache
Query	------ This cache connection is used for the caching of the tag cfquery
Resource	------ This cache connection is used for the Ram Resource (ram://...)

What just happened?

In order to use a **Cache** store, we needed to create a cache connection for it. The default cache type that you can use with Railo Server is the `RamCache`, which will store any object we create in the server's RAM. Once we created our **Cache**, we are ready to use it in our code. Let's try that out now.

Time for action – using the Cache object

Since we have now created our **Cache** connection, we can start using it. Let's try it out:

1. Under our `<Railo Install>/webroot/Chapter_5/` folder, let's create a file called `Listing5_3.cfm` with the following code:

```
<cfset cachePut('hello','Hello World')>
<cfoutput> #cacheGet( 'hello')#</cfoutput>
```

This code puts the string `Hello world` into the cache under the key `hello`. We can then get the item that is present using the `cacheGet()` function and display it.

This code would output:

Hello World

2. We can expand on this example a little, so you get an idea of the other parameters that we can use with the `cachePut()` function:

```
<cfset cachePut('hello','Hello World',createTimespan(0,0,0,10)
  ,createTimespan(0,0,0,10),'mycache')>
<cfoutput> #cacheGet( 'hello', true, 'myCache')#</cfoutput>
```

In this example, we call the `cachePut()` method with the following parameters:

- `KeyName`: This is the key name of our item in the cache.
- `Value`: This is the value we are storing in the cache that we will retrieve with `cacheGet()` later on.
- `LifeSpan`: This is the third argument that defines how long an item will live in the cache; in our example, we do this by using the `createTimeSpan()` function to define a lifespan for `10` seconds.
- `idleTime`: This is the fourth argument that defines how long an item will stay in the cache before being removed, if it is not accessed. In our example, if it is not accessed for `10` seconds, the item will be deleted.
- `CacheConnection`: This is the final parameter that defines the name of the cache connection we are going to be using to store this item.

In the `cacheGet()` function, we now have the following items:

- `KeyName`: This is the name of the key whose value we want to get.
- `ThrowError`: It will have a Boolean value to say that we will throw an error if the item doesn't exist.
- `CacheConnection`: This is the final parameter that defines the name of the cache connection we are going to be using to retrieve this item.

What just happened?

Using the Railo Caching functions is pretty easy, with `cacheGet()` and `cachePut()` being our main entry points. In the next section, we shall see more functions that can be used to introspect the cache.

Time for action – getting well versed with more caching functions

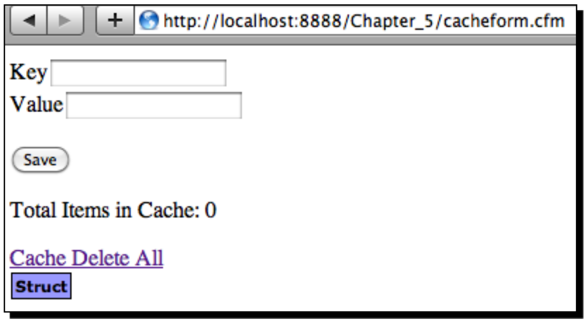
So far, you have seen some simple examples. In the following example, we are going to make a form to which we can add items to our cache, see what is in the whole cache, and then delete individual items as well as the whole cache. Let's get started:

1. In the `<Railo Install>/webroot/Chapter_5` folder, let's create a file called `cacheform.cfm` and add the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>cacheform</title>
  </head>
  <body>
    <cfparam name="FORM.key" default="">
    <cfparam name="FORM.value" default="">
    <cfparam name="URL.delete" default="">
    <cfparam name="URL.deleteall" default="false">
    <form action="cacheform.cfm" method="post">
      <label for="key">Key</label>
      <input type="text" name="key"><br>
      <label for="value">Value</label>
      <input type="text" name="value"><br>
      <p><input type="submit" value="Save"></p>
    </form>
    <cfif Len(FORM.key)>
      <cfset cachePut(FORM.KEY,FORM.VALUE)>
    </cfif>
    <cfif Len(URL.delete)>
      <cfset cacheDelete(URL.delete)>
    </cfif>
    <cfif URL.deleteall>
      <cfset cacheClear()>
    </cfif>
    <cfoutput>
      Total Items in Cache: #cacheCount()# <br>
```

```
        <ul>
          <cfloop array="#cacheGetAllIds()#" index="c">
            <li>#c# : #cacheGet(c)#
              <a href="cacheform.cfm?delete=#c#">Delete</a>
            </li>
          </cfloop>
        </ul>
        <a href="cacheform.cfm?deleteall=true">Cache Delete All</a>
      </cfoutput>
      <cfdump var="#cacheGetAll()#">
    </body>
</html>
```

2. When we call the template in our browser, by going to `http://localhost:8888/Chapter_5/cacheform.cfm`, we should get a form to enter our **Key** and **Value** to the cache:



Total Items in Cache: 0

3. Enter a **Key** with the name `item1` and **Value** of `Item One` and submit the form. You should now see that the **Total Items in Cache** increases to `1`, and you now have your item displayed in the cache. How did this happen? When we submitted our values, it called the following code:

```
<cfif Len(FORM.key)>
  <cfset cachePut(FORM.KEY,FORM.VALUE)>
</cfif>
```

This code simply checks that there is a key defined in the `FORM` scope and then adds that item to the cache; we have seen this before. (Did you notice the `<cfparam>` code at the top make sure that we had already defined our values?)

4. Now that we have an item in our cache, we can go and get it anytime by using `cacheGet()`, but you already knew that right? How about getting a count of all the items in the cache? Well, the following code gets the number of items:

```
Total Items in Cache: #cacheCount()# <br>
```

5. How about looping through the items in our cache? That is easy! We can use the `cacheGetAllIds()` to get an array of IDs and then loop through them while getting each one:

```
<ul>
  <cfloop array="#cacheGetAllIds()#" index="c">
    <li>#c# : #cacheGet(c)#
      <a href="cacheform.cfm?delete=#c#">Delete</a>
    </li>
  </cfloop>
</ul>
```

6. We can get a structure with all the items in our cache simply by calling `cacheGetAll()`.

7. How about deleting items in our cache? As you can see in the previous code, we have a link that passes the key name back to our template, which then checks for the `URL.delete` variable and deletes that key in our cache:

```
<cfif Len(URL.delete)>
  <cfset cacheDelete(URL.delete)>
</cfif>
```

8. We can also delete all the items in our cache by using the `cacheClear()` function.

What just happened?

Using a template, we have managed to add items to our cache and list them. You should also notice that once you put an item in the cache and go back to the page, the items will stay in the cache.

Now that we have looked at the basic cache functions and functionality, we can have a look at what other things the cache can be used for.

Cache providers

Using the RamCache is fine, but RAM is a precious resource. Railo Server allows you to also use a number of external caches. Similar to databases, Railo Server supports different types of cache connections. Imagine you could only use MySQL in Railo Server. This would be a huge restriction, as you might want to harness some of the features of other database types.

We believe that the same is true for caches. Therefore, Railo Server is not limited to one single caching system, but has left the interface open for several cache types you can use, depending on your requirements.

Currently, Railo Server supports the following cache types:

- **RamCache**: This cache is shipped with Railo and is based in memory. The cache is very fast and well suited for small applications, but very quickly pushes its limits.
- **EHCache Lite** `http://ehcache.org/`: This cache is packaged with Railo version 3.2. This cache is also used in other CFML engines in the same way, and it provides a variety of ways to define for how long objects should live and where and when they are stored.
- **EHCache** `http://ehcache.org/` **(Extension)**: This cache works similar to "EHCache Lite", but in addition, it allows the configuration of a cluster by connecting to other EHCache servers through a peer-2-peer cluster.
- **Memcached** `http://memcached.org/` **(Extension)**: This cache is offered as a free extension and works in the same way as "EHCache Lite" does. Just like "EHCache", this cache provides a cluster solution, but not as a peer-2-peer model, but as a client-server model. The data is not stored locally, but on a centralized server, similar to a database.
- **Infinispan** `http://www.jboss.org/infinispan` **(Extension)**: Infinispan is an extremely scalable and highly available data grid platform – 100 percent open source and written in Java. The purpose of Infinispan is to expose a data structure that is highly concurrent, designed ground-up to make the most of modern multi-processor/multi-core architectures, while at the same time providing distributed caching capabilities. It is also optionally backed by a peer-to-peer network architecture to distribute state efficiently around a data grid.
- **Membase** `http://www.membase.org` **(Extension)**: Membase is a distributed key-value database management system, optimized for storing data behind interactive web applications. Membase automatically spreads data and I/O across servers. This "scale out" approach at the data layer permits virtually unlimited growth of transaction capacity, with linear increases in cost and constant per-operation performance.
- **CouchDB** `http://couchdb.apache.org/` **(Extension)**: Apache CouchDB is a document-oriented database that can be queried and indexed in a MapReduce fashion using JavaScript. CouchDB also offers incremental replication with bi-directional conflict detection and resolution.

As you can see, Railo Server supports a number of cache types and the list is growing. Of course, you can also write a cache driver for your favorite caching system too, as you will see in *Chapter 9*, *Extending Railo Server*.

These can be installed through the extension providers, which we will go into more detail in *Chapter 9*, *Extending Railo Server*.

Cache types

So far, we have looked at using one type of cache, the object cache, using the RamCache connection. This is not the only functionality of caches in Railo Server. They can be used for caching other types of resources too.

Railo Caches can be used with other services and functions, such as queries, templates, objects, and resources.

Template cache

The tag `<cfcache>` has the actions `flush`, `get`, and `put` in order to read and write objects, similar to the functions `CacheGet`, `CachePut`, and `CacheClear`. You can also use this tag for caching templates of your application for a period of time.

Time for action – caching a page with cfcache

Let's see how the `<cfcache>` tag can cache a template for us:

1. In our `<Railo install>/webroot/Chapter_5/` folder, let's create a template called `templatecache.cfm` and add the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Template Cache</title>
  </head>
  <body>
    <cfoutput>
      Current Time: #TimeFormat(Now(), "HH:mm:ss")#
    </cfoutput>
  </body>
</html>
```

2. When you run this template by going to `http://localhost:8888/Chapter_5/templatecache.cfm`, you should see the current time displayed, for example, `Current Time: 15:02:25`. Each time you refresh this, it will show the current time and stay up-to-date.

3. Let's add some code to cache this whole page. At the top of the page, add the following code:

```
<cfcache action="cache">
<!DOCTYPE html>
<html>
  <head>
```

```
    <title>Template Cache</title>
  </head>
  <body>
    <cfoutput>
      Current Time: #TimeFormat(Now(), "HH:mm:ss")#
    </cfoutput>
  </body>
</html>
```

The `<cfcache action="cache">` tag at the top of our page will cache the whole page.

4. Now, when we reload the page each time, it will keep displaying the original time. This is because the page is now cached to disk. You can see this if you go to the `<Railo install>/webroot/WEB-INF/railo/cache` folder. There is a file in there called `31d4007e8254d94f98704ffeb17b8243.cache` (the name might vary based on your system).

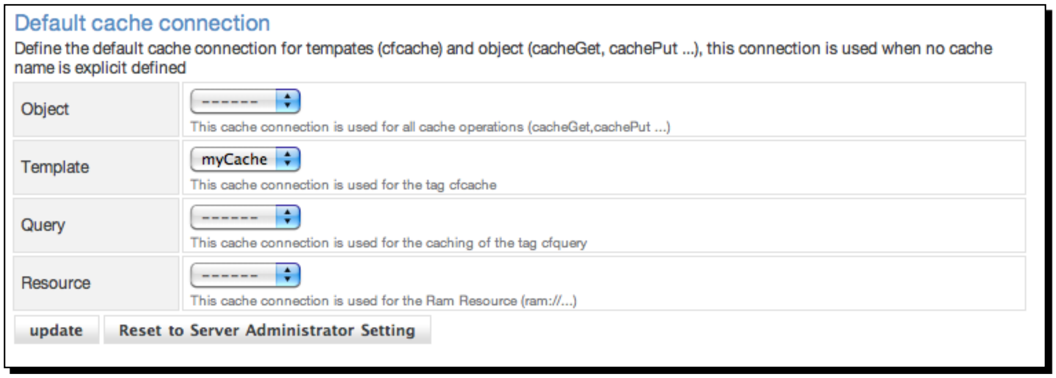
5. What happens if we want to invalidate the cache? We have a few options; we could delete the `31d4007e8254d94f98704ffeb17b8243.cache` file from the hard drive or we could use the `<cfcache>` tag again. Let's use the tag and add the following code to the top of our template:

```
<cfif isDefined("URL.flush")>
<cfcache action="flush">
</cfif>
<cfcache action="cache">
<!DOCTYPE html>
<html>
  <head>
    <title>Template Cache</title>
  </head>
  <body>
    <cfoutput>
      Current Time: #TimeFormat(Now(), "HH:mm:ss")#
    </cfoutput>
  </body>
</html>
```

When we run our template again, it will still be cached until we add a `URL` parameter to flush the cache. If you run the URL `http://localhost:8888/Chapter_5/templatecache.cfm?flush`, you will be able to see that the template is removed from the cache.

6. The template cache is still reading from the filesystem, and you can imagine with a lot of templates this cache could become slow. We can change it to use the URL cache that we created earlier by changing what cache the templates are using. Let's head back to the Railo Web Administrator and set a default cache by going to `http://localhost:8888/railo-context/admin/web.cfm` and then clicking on the **Cache** link under **Services**.

7. In the **Default cache connection** section, select the **myCache** connection next to the **Template** cache and set all the other types to the blank connection:



Default cache connection

Define the default cache connection for tempates (cfcache) and object (cacheGet, cachePut ...), this connection is used when no cache name is explicit defined

Object	------ This cache connection is used for all cache operations (cacheGet,cachePut ...)
Template	myCache This cache connection is used for the tag cfcache
Query	------ This cache connection is used for the caching of the tag cfquery
Resource	------ This cache connection is used for the Ram Resource (ram://...)

8. Then click **update**.

9. Now all the templates will be cached in the RAM instead of the hard drive.

What just happened?

In the previous example, we managed to cache a whole template using the `<cfcache>` tag. We saw a couple of actions that the `<cfcache>` tag provides to both add items to a cache as well as be able to flush the contents of the cache. We also saw that we can assign a cache connection to the template resources, so that they are also cached to any type of cache provider we choose.

Partial template caching

In the previous section, we saw how we cached a template (in fact, a whole template, as we could have included other templates within our main template), which is very useful, but what if we only want to cache a part of a page?

The `<cfcache>` tag can also allow you to cache a portion of the page, so let's try this out.

Time for action – caching content within a template

Using our previous template, `templatecache.cfm`, let's add some caching to only a part of our template:

1. Open up the `templatecache.cfm` file and replace the contents with the following block of code:

```
<cfif isDefined("URL.flush")>
  <cfcache action="flush">
</cfif>
<!DOCTYPE html>
<html>
  <head>
    <title>Template Cache</title>
  </head>
  <body>
    <cfoutput>
    <cfcache action="content" key="myCachedTime">
      Cached Time: #TimeFormat(Now(), "HH:mm:ss")#
    </cfcache>
      Current Time: #TimeFormat(Now(), "HH:mm:ss")#
    </cfoutput>
  </body>
</html>
```

2. As you can see, we removed the `<cfcache action="cache">` from the top of our template. This removes caching for the whole template. We have now added a `<cfcache action="content">` around the output of time, and also added the current time, so that we can see the difference.

3. When you load up this template and refresh it a few times by going to `http://localhost:8888/chapter_5/templatecache.cfm`, you can see that the time differs as the content inside the `<cfcache>` tag will now be stored, but the rest of the template will run as normal:

```
Cached Time: 15:43:00 Current Time: 16:14:10
```

What just happened?

We have managed to cache only a portion of our page, maybe a part that would contain a lot of processing that normally wouldn't need to be processed for each request. This greatly improves our template speeds and how the user perceives our application.

Query cache

You can use the `<cfquery>` tag to retrieve records from a database, but what happens if this data doesn't change very often?

Railo Server can cache database requests for a period of time, using the `cachedwithin` and `cachedafter` attributes, which define a time span of how long a query will be held in memory.

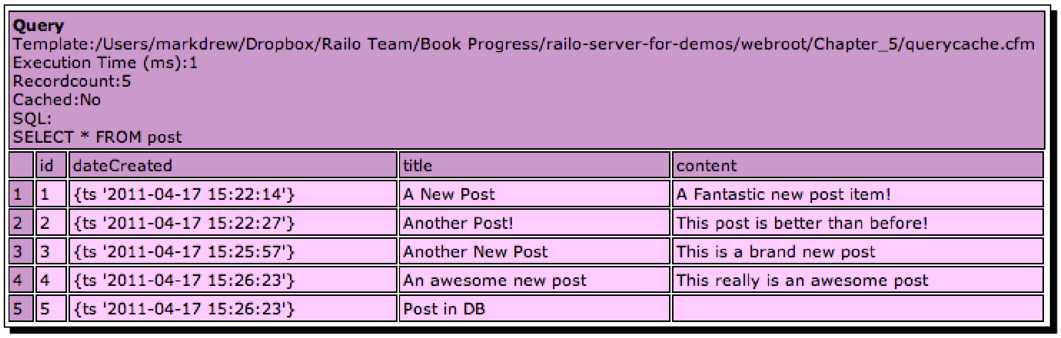
Time for action – caching a query using cachedwithin

Let's say, we want to display a list of our blog posts. Since we don't want to query the database all the time, we are going to cache our query for a set length of time. Let's first get the items from our database:

1. In your `<Railo Install>/webroot/Chapter_5` folder, create a template called `querycache.cfm` with a simple query and a dump of the results:

```
<cfquery name="getPosts" datasource="railoblog">
  SELECT * FROM post
</cfquery>
<cfdump var="#getPosts#">
```

The output of this query should look something like the following screenshot:



Query
Template:/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/Chapter_5/querycache.cfm
Execution Time (ms):1
Recordcount:5
Cached:No
SQL:
SELECT * FROM post

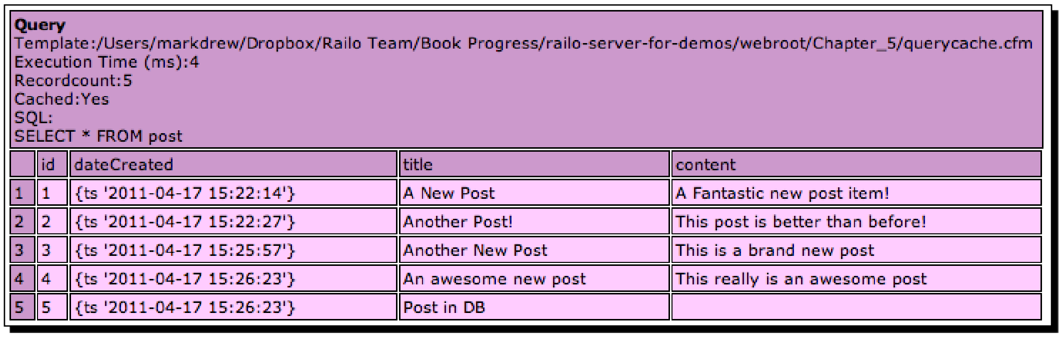
	id	dateCreated	title	content
1	1	{ts '2011-04-17 15:22:14'}	A New Post	A Fantastic new post item!
2	2	{ts '2011-04-17 15:22:27'}	Another Post!	This post is better than before!
3	3	{ts '2011-04-17 15:25:57'}	Another New Post	This is a brand new post
4	4	{ts '2011-04-17 15:26:23'}	An awesome new post	This really is an awesome post
5	5	{ts '2011-04-17 15:26:23'}	Post in DB	

2. At the top of the query, you can see that it says **Cached: No**

3. Let's add a `cachedwithin` attribute to the `<cfquery>` tag:

```
<cfquery name="getPosts" datasource="railoblog"
  cachedwithin="#CreateTimeSpan(0,0,5,0)#">
  SELECT * FROM post
</cfquery>
<cfdump var="#getPosts#">
```

When we now run the template a couple of times, we will get the following result:



Query
Template:/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/Chapter_5/querycache.cfm
Execution Time (ms):4
Recordcount:5
Cached:Yes
SQL:
SELECT * FROM post

	id	dateCreated	title	content
1	1	{ts '2011-04-17 15:22:14'}	A New Post	A Fantastic new post item!
2	2	{ts '2011-04-17 15:22:27'}	Another Post!	This post is better than before!
3	3	{ts '2011-04-17 15:25:57'}	Another New Post	This is a brand new post
4	4	{ts '2011-04-17 15:26:23'}	An awesome new post	This really is an awesome post
5	5	{ts '2011-04-17 15:26:23'}	Post in DB	

You can see that the query now has **Cached: Yes**

What just happened?

By adding the `cachedwithin` attribute to our `<cfquery>` tag, we are able to cache the results of a query for a period of time. The `cachedwithin` attribute takes a variable returned from the `CreateTimeSpan()` function, which takes four arguments: `Days`, `Hours`, `Minutes`, and `Seconds`. The `<cfquery>` tag now won't run against the database, but use the cached results for a period of time (`5` minutes, as we defined in our `CreateTimeSpan()` function).

Resource cache

Railo Server has the ability to write and read templates straight from RAM as well as other resources. We shall have a closer look at this functionality in *Chapter 8*, *Railo Resources*, but for now, you should know that this is possible.

For example, you can run the following code to write a variable to a file in RAM called `susi.txt`:

```
<cffile action="write" file="ram://susi.txt" output="Hello Susi">
```

This functionality is limited, because the memory itself is a precious resource that is shared by all applications on the server. Also, any items you have saved there will be flushed if the server is restarted.

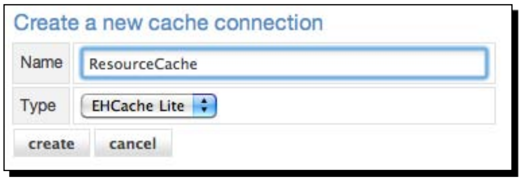
Railo Server has the ability to set which cache provider will be used for our resources. We can assign a better caching system rather than the default RAM cache.

As with the template cache, we can create a new cache connection to, for example, EHCache Lite, and assign that to our resource cache.

Time for action – assigning an EHCache Lite connection to resources

Since we want to provide a better cache provider for our resources, we can now go and create an EHCache Lite connection and assign it as the default cache provider for our resources:

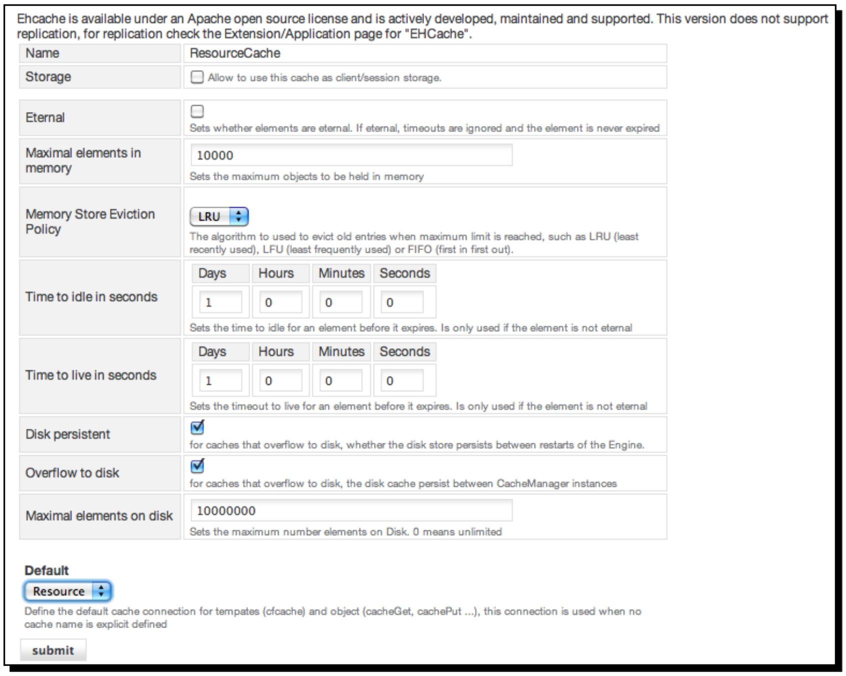
1. Go to the Railo Web Administrator through `http://localhost:8888/railo-context/admin/web.cfm` and log in, if required.
2. Click on the **Cache** link under the **Services** section to get a list of our existing caches.
3. Create a new cache connection called `ResourceCache` of the type **EHCache Lite**:



Create a new cache connection

Name	ResourceCache
Type	EHCache Lite

4. Once we have **clicked** on create, we have a number of options (they are fine set as default for the moment). Make sure that the **Default** drop-down is set to **Resource**:



Ehcache is available under an Apache open source license and is actively developed, maintained and supported. This version does not support replication, for replication check the Extension/Application page for "EHCache".

Name	ResourceCache
Storage	□ Allow to use this cache as client/session storage.
Eternal	□ Sets whether elements are eternal. If eternal, timeouts are ignored and the element is never expired
Maximal elements in memory	10000 Sets the maximum objects to be held in memory
Memory Store Eviction Policy	LRU The algorithm to used to evict old entries when maximum limit is reached, such as LRU (least recently used), LFU (least frequently used) or FIFO (first in first out).
Time to idle in seconds	Days 1, Hours 0, Minutes 0, Seconds 0 Sets the time to idle for an element before it expires. Is only used if the element is not eternal
Time to live in seconds	Days 1, Hours 0, Minutes 0, Seconds 0 Sets the timeout to live for an element before it expires. Is only used if the element is not eternal
Disk persistent	☑ for caches that overflow to disk, whether the disk store persists between restarts of the Engine.
Overflow to disk	☑ for caches that overflow to disk, the disk cache persist between CacheManager instances
Maximal elements on disk	10000000 Sets the maximum number elements on Disk. 0 means unlimited

Default

Resource

Define the default cache connection for tempates (cfcache) and object (cacheGet, cachePut ...), this connection is used when no cache name is explicit defined

5. Now when we run the previous code, all our calls to the `ram://` resource will be stored using the **EHCache Lite** connection that we have now defined.

What just happened?

By default, the `ram://` resource is stored in the server's RAM memory. By creating a new cache provider to the included **EHCache Lite** service, we can now store resources in a more permanent, yet fast, caching system.

Summary

This has been a varied chapter that has introduced you to the Application Lifecycle of Railo Applications, ORM capabilities, and caching services available with Railo Server.

You should now be able to:

- Manage different parts of your requests, sessions, and application lifecyle using the `Application.cfc` file
- Create mappings between Railo Components and database tables to persist them in a database
- Create different caches to store variables and templates for faster retrieval
- Cache queries to reduce database lookups for data that doesn't change frequently

Since we have now got a good grounding in the services of Railo Server and got used to coding with CFML tags, in the next chapter, we are going to explore the other way of coding in CFML, without using tags. We are also going to investigate some of the built-in objects available in Railo Server.

6

Advanced CFML Functionality

So far we have looked at how we can write CFML with tags and functions. This is a great way to develop web applications since it fits well with the way web pages are built, that is, through HTML.

In this chapter we are going to look at the alternative ways to write CFML language, namely, by using a scripting language called CFScript.

In this chapter, we will cover:

- Scripting formats that are available in Railo Server
- How to leverage CFScript with your code
- The in-built components in Railo Server

By the end of the chapter, we will be able to write applications using the CFScript syntax which accesses outside resources without the need to use tags.

Let's dive right in!

Scripting within Railo Server

So far all the examples and code you have written have used a tag-based format. Even when we used functions that had no output, we used the `<cfset>` tag. It's a natural way to program when developing web applications because the output language is HTML, which is also a tag-based language. But, is this the only way to do this? Let's have a look at the reason for there being another way to do things and how it could improve our current situation.

Let's have a look at the pros and cons of using these tags in the following two sections.

Why tags are good

Using a tag-based format has many advantages. Primarily, it makes for a very easy learning curve for the user. Although there are many tags in Railo Server (about 126 on the last count), the syntax remains the same:

```
<cfTAGNAME ATTRIBUTE1="VALUE" ...>
```

When using a self-closing tag, we use the following:

```
<cfTAGNAME ATTRIBUTE1="VALUE"/>
```

In case of tags that wrap some content in the format, it is as follows:

```
  <cfTAGNAME ATTRIBUTE1="VALUE" ...>

  </cfTAGNAME>
```

Therefore, it is easy to figure out what tag you might want to use. Because Railo Server is not overly strict on syntax, you don't have to close the single tags if you don't want to (that is, adding /> at the end). This makes it a very forgiving format to learn the language—somewhat similar to HTML

Also, with this format, you are also able to parse the CFML rather easily, so different tools, like text editors, can work with it easily.

The only change to this syntax is the use of the hash mark **#** (also known as the pound sign) to surround variables that need to be outputted to the browser.

This makes the whole language fairly consistent in this format, especially when mixed with HTML, as shown in the following lines of code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>cacheform</title>
  </head>
<body>
  <cfquery name="getList" datasource="raniloblog">
  SELECT * FROM posts
  </cfquery>
    <ul>
      <cfloop query="#getList#">
      <li>
      <cfoutput>#title#</cfoutput>
getList.      </li>
      </cfloop>
    </ul>
</body>
</html>
```

As you can see in the preceding code, CFML mixes quite well with HTML; it's readable and you can tell which are CFML tags and which are HTML rendering tags.

Why tags are bad

The benefits of CFML formatting can also be its detriment. The logic in code isn't always as simple as one tag; it could be a number of statements, settings and parsing variables, calling functions, and any number of other features of the language. You could end up with something that, in tag format, is actually too verbose.

If you are coding business logic with nothing to be outputted to the browser, it may be better to place that code in a component.

Let's have a look at some code; it doesn't matter what it does ultimately, but it's to give you an idea of where CFML tags can actually get in the way:

```
<cfset selectedDir = expandPath("../../../Chapter 6/images")>
<cfset aFiles = DirectoryList(absolute_path=selectedDir, filter="*.
png", listInfo="name")>
<cfset prefix = "3401_05">
<cfset imageCountNumber = 0>
<cfloop array="#aFiles#" index="f">
  <cfif f.startsWith(prefix)>
      <!--- increase the count --->
    <cfset FoundNumber = ListLast(ListFirst(f,"."), "_")>

    <cfif FoundNumber GTE imageCountNumber>
      <cfset imageCountNumber = FoundNumber + 1>
    </cfif>
  </cfif>
</cfloop>


<!--- rename the image files to the proper format --->
<cfloop array="#aFiles#" index="f">
  <cfif NOT f.startsWith(prefix)>
    <cfset newName = prefix & "_" & NumberFormat(imageCountNumber,
"00") & "." & ListLast(f,".")>
    <cfset FileMove("#selectedDir#/#f#","#selectedDir#/#newName#")>
    <cfset imageCountNumber++>
  </cfif>
</cfloop>
```

In the previous code, we set a number of variables with the `<cfset>` tag (in fact, we have done that nine times already!), and in some cases, we aren't even setting anything; we are just calling it to call a function to do some action.

More importantly, because the code is basically about renaming a file, there is no output to the user, so the benefits of a tag-based language seem to be diminished. It seems a bit superfluous to have all those `<cfset>` tags too.

Let's have a look at another way in which we could write the previous code:

```
selectedDir = expandPath("../../../Chapter 6/images");
aFiles = DirectoryList(absolute_path=selectedDir, filter="*.png",
listInfo="name");
prefix = "3401_05";
imageCountNumber = 0;
for(f in aFiles){
  if(f.startsWith(prefix)){
     FoundNumber = ListLast(ListFirst(f,"."), "_");
    if(FoundNumber >= imageCountNumber){
      imageCountNumber = FoundNumber + 1;
    }
  }
}

for(f in aFiles){
  if(!f.startsWith(prefix)){
     newName = prefix & "_" & NumberFormat(imageCountNumber, "00") &
"." & ListLast(f,".");
     FileMove("#selectedDir#/#f#","#selectedDir#/#newName#");
    imageCountNumber++;
  }
}
```

What we have now is something that is more concise, without the syntactic noise that tags can bring. We are still writing CFML and it seems we have just removed the tags (apart for the "for loop") and everything is pretty much the same, but there is less "noise".

Let's see how we can use this to our advantage.

The <cfscript> tag

The reduction of code we saw in the previous code example was achieved using the other language syntax that Railo Server supports. This syntax is known as CFScript.

Similar to the `<SCRIPT>` tag in HTML, we are able to use the `<cfscript>` tag in our CFML templates to use the CFScript syntax. This is very similar to `ECMAScript` (or JavaScript) in its general notation.

You can think of it as CFML, but without the `<cf` at the start of a tag and the `>` at the end. Or another way of looking at it is that we are calling the `<cfset>` tag without having to use the `<cfset>` tag.

Let's look at some of the differences between the tag-based CFML language and its CFSC counterpart.

Loops

Loops are used for many things in Railo, but in the tag-based CFML language, the main tag for all of the looping interactions is the `<cfloop>` tag.

Looping lists

Lists are the simplest type of data structures that you can have in CFML, it's basically a long string delimited by some character, usually a comma (",").

Time for action – looping through a list

Let's loop through a list using `<cfloop>` and then compare it to looping through it using `<cfscript>`:

1. First, create a file named `looplist.cfm` in the `<Railo Installation Directory>/webroot/Chapter_6` folder and create a list by adding the following code:

```
<cfset myList = "Item One,Item Two,Item Three">
```

2. Now, let's loop through the list using `<cfloop>` by appending the following code:

```
<cfloop list="#myList#" index="i">
  <cfoutput>#i#<br></cfoutput>
</cfloop>
```

3. This prints out the following output, as expected:

```
Item One
Item Two
Item Three
```

4. Let's replace the `<cfloop>` with the following lines of code:

   ```
   <cfscript>
     for(i = 1; i <= ListLen(myList); i++){
       WriteOutput(ListGetAt(myList, i) & "<br>");
     }
   </cfscript>
   ```

5. We get the same output, but of course, we seem to be doing it rather differently.

What just happened?

The `<cfloop>` tag is incredibly powerful because it knows how to loop over a number of objects including files. In the previous example, we use the `for(){}` loop to iterate from 1 whilst the variable `i` is less than or equal to the length of the list. This is not the shortest example, but it does show that you can loop through a list and then use the `ListGetAt()` built-in function to get the item there.

Let's look at more complex examples now.

Looping arrays

Arrays are actually easier to loop through in CFScript than lists. Let's just go with an example to see how easy they are.

Time for action – looping an array

1. This time, let's create another template named `looparray.cfm` in the same directory as we created the `looplist.cfm` template.

2. Let's create an array by putting the following code at the top of our template:

   ```
   <cfset myArr = ["Item One", "Item Two", "Item Three"]>
   ```

3. Now let's loop through it using `<cfloop>`; we do this by adding the arguments `array` and `index`:

   ```
   <cfloop array="#myArr#" index="a">
     <cfoutput>#a#<br></cfoutput>
   </cfloop>
   ```

4. We passed the myArr variable into the array attribute, and to display the contents of each item, we output #a#. Now, let's do this using CFScript. Let's add the following code:

   ```
   <cfscript>
     for(a in myArr){
       WriteOutput(a & "<br>");
     }
   </cfscript>
   ```

5. Now when we run the template again, we should see the same output as with <cfloop>, but the for loop is much more contained.

What just happened?

When looping arrays, the <cfscript> syntax is much more refined. It makes sense when you say it out in English, "For (every item called) a in myArr (do this)."

Looping structures

Structures in CFML are a map of key/value pairs, and as you might have guessed by now, <cfloop> also has attributes designed to be used with looping through a structure. Let's see how this works in tags and <cfscript>.

Time for action – looping through a structure

1. In the same directory where we placed our other templates, let's create a template named loopstruct.cfm and let's add the following code that creates a simple structure that we can loop over:

   ```
   <cfset myStruct = {item1="Item One", item2="Item Two", item3="Item
   Three"}>
   ```

2. The previous code has a number of keys (item1, item2, item3) that have values; let's loop through them with <cfloop> by using the collection and item attributes:

   ```
   <cfloop collection="#myStruct#" item="s">
     <cfoutput>#s# = #myStruct[s]# <br></cfoutput>
   </cfloop>
   ```

3. This code would output the following values:

   ```
   ITEM3 = Item Three
   ITEM2 = Item Two
   ITEM1 = Item One
   ```

4. The variable `s` refers to the current key that we are looping over, and to get the content, we can use `#myStruct[s]#`

5. Let's try this with `<CFSCRIPT>`. Now and see the difference:

   ```
   <cfscript>
     for(s in myStruct){
       WriteOutput(s & " = " & myStruct[s] & "<br>");
     }
   </cfscript>
   ```

6. The previous code would also output the same values as the `<cfloop>`.

What just happened?

We can see that the syntax for the `<cfscript>` version of structure loop is identical to the array version of the loop. If we ignore the output (we don't always output things when we are looping over items), it is a much tidier syntax and less verbose. This is where `<cfscript>` really shines!

Looping queries

As we have seen a number of times so far, database queries are very well handled within Railo Server and the CFML language. Let's see how we can handle them with `<cfscript>` too.

Time for action – looping over queries

1. In the same folder that we have been placing all of our examples so far, let's create a template named `loopquery.cfm` and put a `<cfquery>` statement at the top:

   ```
   <cfquery name="qItems" datasource="railobook">
   SELECT * FROM Users
   </cfquery>
   ```

2. Now, we can loop through the results using `<cfloop>` and the `query` attribute as follows (we have seen this a number of times now):

   ```
   <cfloop query="qItems">
     <cfoutput>#qItems.username#<br></cfoutput>
   </cfloop>
   ```

3. Nothing new here; we just get a listing of our users in the database. Now, let's try it with `<cfscript>`:

```
<cfscript>
  for(q in qItems){
    WriteOutput(qItems[q]["username"] & "<br>");
  }
</cfscript>
```

4. When we run this code, we get an error!—**key [username] not found**. This is because each item in a query is actually the column, not the row! Let's change our code to loop through all the rows:

```
<cfscript>
   for(r=1; r LTE qItems.recordcount; r++){
       WriteOutput(qItems["username"][r] & "<br>");
   }
 </cfscript>
```

5. Now, we get a list of users as we would expect!

What just happened?

Even though we expected to use the `for(q in qItems)` syntax to loop over a query, things are slightly different when you think about the structure of a query. It is still easy to loop over it in `<cfscript>`, but we need to remember that we have to loop over all the rows first before outputting the column (either directly by name, or by having another loop inside the row loop).

Scripted components

Components are a great location to place the business logic of your application into. They are object-oriented and allow you to separate your business logic from your display logic and build the foundation for creating maintainable CFML applications. Despite this, they can get rather long to code. Let's look at a simple component that we might use in an application. The `Person` component will only have two properties, `name` and `age`, and we will have functions (called getters) to get the values of the `name` and `age` properties.

Time for action – creating the component

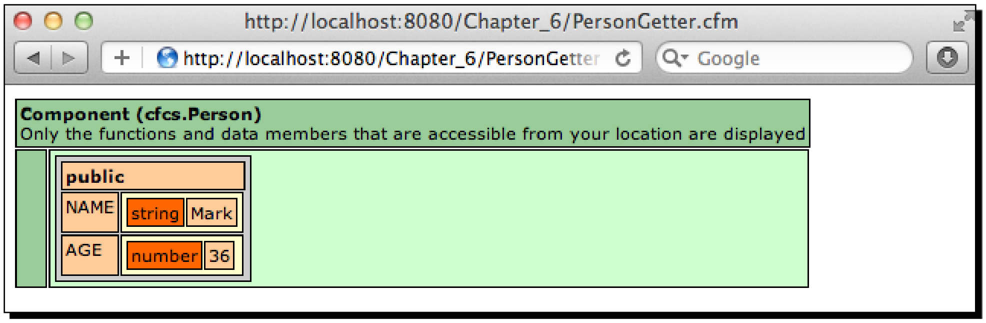
Let's start off by creating a folder to store our component. Under the `<Railo Install Directory>/webroot`, let's create a folder named `cfcs`.

1. In the `cfcs` directory, create a template named `Person.cfc` and put the following code inside it:

```
<cfcomponent output="false">
  <cfset this.name = "">
  <cfset this.age = "">
</cfcomponent>
```

2. This is the simplest form of a component. Let's create another template under `<Railo Install Directory>/webroot/Chapter_6/` named `PersonGetter.cfm`; this template will be used to create an instance of the `Person.cfc` object and populate it by adding the following code:

```
<cfset person = new cfcs.Person()>
<cfset person.name = "Mark">
<cfset person.age = 36>
<cfdump var="#person#">
```


http://localhost:8080/Chapter_6/PersonGetter.cfm
http://localhost:8080/Chapter_6/PersonGetter
Google

```
Component (cfcs.Person)
Only the functions and data members that are accessible from your location are displayed
public
NAME   string  Mark
AGE    number  36
```

3. The `this` scope in a component is actually public. This means it can be read and modified from outside the component (as we can see in the previous code listing), which is not best practice. Let's make the properties private and add getters and setters:

```
<cfcomponent output="false">
  <cfset variables.name = "">
  <cfset variables.age = "">

  <cffunction name="getName" output="false" returntype="string">
    <cfreturn variables.name>
  </cffunction>
  <cffunction name="setName" output="false" returntype="void">
    <cfargument name="name" type="String" required="true">
    <cfset variables.age = arguments.age>
  </cffunction>
</cfcomponent>
```

4. In the previous code, we have changed the `this` scope to the private `variables` scope within the component. We have also added a number of `<cffunction>` tags to add methods to our component that set and get the variables. Now, we need to change the `PersonGetter.cfm` file to use these methods:

```
<cfset person = new cfcs.Person()>
<cfset person.setName("Mark")>
<cfset person.setAge(36)>
<cfoutput>#person.getName()# - #person.getAge()#</cfoutput>
```

5. Even though this is now a better practice, it takes a bit of coding to get all of what's written out. This is where `<cfscript>` can help. Let's create another component that will behave the same way, but written solely in CFScript.

6. In the `cfcs` folder, create a template named `PersonScript.cfc`. Notice that it is still a `.cfc` file. Let's add the following code:

```
component {
  variables.name = "";
  variables.age = "";

  String function getName(){
    return variables.name;
  }

  function setName(String required  name){
    variables.name = arguments.name;
  }

  Numeric function getAge(){
    return variables.age;
  }

  function setAge(Numeric required age){
    variables.age = arguments.age;
  }

}
```

7. Wow! This seems more condensed right? There is less syntactic noise from the tags and yet we are keeping the functionality the same. Also, because it is CFScript, we don't have to keep putting `output="false"` in our code. Using the tag-based component functions, we need `output="false"` if we want to prevent any output. Of course, we could output code straight from the function if we want, although this is also not good practice.

8. Now, to test our new component, we need to change one line in the `PersonGetter.cfm` file:

```
<cfset person = new cfcs.PersonScript()>
<cfset person.setName("Mark")>
<cfset person.setAge(36)>
<cfoutput>#person.getName()# - #person.getAge()#</cfoutput>
```

What just happened?

We created a simple component using standard CFML tags, which works fine, but it adds a lot of noise, especially in simple objects. By modifying it to use the CFScript syntax, we can see that we save a lot of typing, and also, the component is much clearer. This is one of the benefits of using CFScript over tags.

Scripting tags

There are many ways to code something, and now that you have seen the benefits of CFScript, you might want to code everything that way. Therein lies a problem. For example, even though there are over 500 functions you can call, there are also over 100 tags available to you in Railo Server and sometimes the functionality doesn't overlap. So how do we get access to that functionality in CFScript?

A number of tags have the ability to be used from CFScript directly by simply removing the `<cf` at the start and replacing the > at the end with `;`. Let's look at an example.

Time for action – getting the contents of another site

Let's say that in our code we want to get the contents from another site, for example, the HTML content of the `http://www.getrailo.org` website. Let's do this using tag-based code first:

1. Create a template under `<Railo Install Directory>/webroot/Chapter_6/` named `getrailo.cfm` and put the following code inside it:

```
<cfhttp url="http://www.getrailo.org" method="GET">
<cfdump var="#CFHTTP.filecontent#">
```

2. When we run the previous code by going to `http://localhost:8888/Chapter_6/getrailo.cfm`, we get a big dump of the HTML that is hosted at `http://www.getrailo.org`. This is pretty simple, but there isn't a CFScript function that will do this, let's use what we have learned about some tags to see if it works without the `<cf` at the start of the `<cfhttp>` tag:

```
<cfscript>
  http url="http://www.getrailo.org" method="GET";
  dump(CFHTTP.filecontent);
</cfscript>
```

3. We now see exactly the same output we saw with the tag version!

What just happened?

A number of CFML tags in Railo Server can be invoked from the CFScript syntax with little modification as to how they work. All it takes is removing the starting `<CF` and replacing the final `>` with a `;` to get them working. This makes writing complex components using the CFScript syntax much easier!

Scripting wrapped tags

As we have just seen, it's not difficult using a single tag call in CFScript, but what happens with tags that wrap other content? If you remember when we looped over a list, things got tricky, as we had to loop over the length of the list, and then use the `ListGetAt()` function, especially because the tag version is so much simpler to read. How about we try to re-write this, as we now know that we can use tags in CFScript?

Time for action – using the <cfloop> tag in CFScript

1. Let's start off by copying the original list loop that we looked at earlier into a new file named `loopscript.cfm` under the `<Railo Install Directory>/webroot/Chapter_6/` directory:

```
<cfset myList = "Item One, Item Two, Item Three">
<cfloop list="#myList#" index="l">
  <cfoutput>#l#<br></cfoutput>
</cfloop>
```

2. Now, if we head to `http://localhost:8888/Chapter_6/loopscript.cfm`, we should get the list output as follows:

```
Item One
Item Two
Item Three
```

3. Let's change the code to just use CFScript. We will use the `<cfloop>` tag, but now use angular brackets to wrap the content; in other words, we use what's inside the `<cfloop>` tag:

   ```
   <cfscript>
     myList = "Item One, Item Two, Item Three";
     loop list="#myList#" index="l" {
       WriteOutput( l & "<br>");
     }
   </cfscript>
   ```

4. Running this code, we now get exactly the same output!

   ```
   Item One
   Item Two
   Item Three
   ```

What just happened?

Wow! In the previous example, you saw how to use a CFML tag using the CFScript syntax where a lot more code would have been required to do the same thing. This is the power that CFML and Railo Server give you!

Scripting wrapped tags—Part 2

In the previous section, we saw how we could script wrapped tags using the CFScript notation. It doesn't end there of course; there are a number of tags that we can use that actually take child tags. Let's look at a simple example of getting an individual item from a query.

To protect ourselves from nasty SQL injection attacks, we can parameterize variables passed to a query using the `<cfqueryparam>` tag. This ensures that our inputs into a query are what they say they are. Let's look at a simple example.

Time for action – get a user by his/her ID

Let's first write the script as we would have done using a `<cfquery>` tag and then see how we can convert it to the CFScript format.

1. Create a template under the `<Railo Install Directory>/webroot/Chapter_6/` named `queryscript.cfm` and let's put the following code in there:

   ```
   <cfparam name="url.id" type="numeric">
   <cfquery name="getUser" datasource="railobook">
   ```

```
  SELECT * FROM Users WHERE id = <cfqueryparam cfsqltype="cf_sql_
numeric" value="#url.id#">
</cfquery>

<cfoutput>
  #getUser.username#
</cfoutput>
```

2. In the previous code, the first line sets up a parameter called `url.id`; this means that an error will be thrown if an `id` is not passed in the `URL` or if it is passed and is not numeric.

3. We then create our query, but instead of simply passing the variable to the query, we use `<cfqueryparam>` to say that the value we are passing is of a database type `cf_sql_numeric`. Finally, we output the first item we get back from the results of the query. So far so good. Let's rewrite this in the CFScript format.

```
<cfscript>
  param name="url.id" type="numeric";
  query name="getUser" datasource="railobook" {
    WriteOutput("SELECT * FROM Users WHERE id = ");
    queryparam cfsqltype="cf_sql_numeric" value="#url.id#";
  }
  WriteOutput(getUser.username);
</cfscript>
```

What just happened?

In the first line, we have replaced the `<cfparam>` tag for the near identical `param` scripted tag. Then, we replace the `<cfquery>` tag with the `query` scripted tag and open the brackets after it.

Using the `WriteOutput()` function, we write our SQL statement to the `query` scripted tag. Then, wherein the tag version the `<cfqueryparam>` tag would have gone, we have simply added the `queryparam` scripted tag.

As we have seen, tags that have child tags can simply be written inside the curly braces. In some tags where we have a mixture of some text content as well as other tags, we can mix them by placing them in order and using the child tags, as you would have done in the tag versions of those tags.

Built-in components

So far, we have seen that we have some good options for converting tag-based CFML code to using the CFScript syntax. However, as you can see from the previous query example, this could start getting very messy. There would be a lot of `WriteOutput()` functions if you have a number of statements, and especially if you have a number of child parameters that you would need to pass.

Another issue is that if you are writing this code dynamically, there will be a *lot* of concatenation of strings, and eventually, the code will not be readable, which is certainly not the point of Railo Server. Normally, a good way of resolving this would be by abstracting away a lot of the code into a component and then just calling methods on them. Luckily, Railo Server includes a few components already to help you do this.

The Query built-in component

As we saw in our CFScript version of a `<cfquery>`, the code could get rather complex. Luckily, there is a component shipped with Railo Server that allows you to easily call complex tags. The `Query` component in Railo Server is a perfect example of this, and the best way to understand it is to take it for a test drive.

Time for action – using the Query component

Let's redo the code in our `queryscript.cfm` file to use the `Query` component:

1. Delete all the code in `<Railo Install Directory>/webroot/Chapter_6/queryscript.cfm` and replace it with a blank `<cfscript>` tag and a call to a new `Query` object:

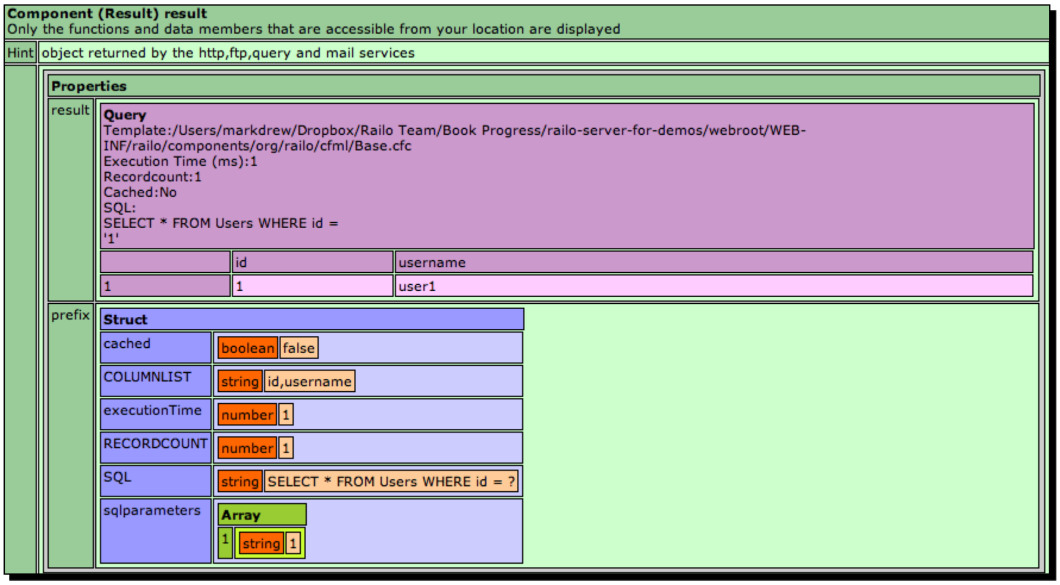
```
<cfscript>
  myQuery = new Query(datasource="railobook");
  myQuery.setSQL("SELECT * FROM Users WHERE id = :id");
  myQuery.addParam(name="id", type="cf_sql_numeric", value=url.
id);
  myResult = myQuery.execute();
  dump(myResult);
</cfscript>
```

2. In the first line, we can call our `new Query()` object and we pass it a `datasource`.

3. We then call the `setSQL()` method, which you might notice has the SQL for the query, but more importantly, it has `:id`. This is the name of the parameter we are going to replace.

4. We then called the `addParam()` method on the `Query` object to add a `queryparam`, passing in the name (which will replace the `:id` variable in the SQL statement), the type, and the value that we get from the `URL` scope.

5. Finally, we call the `myQuery.execute()` method; this actually runs the query against the database and it returns a result object, which, when dumped, looks like this:



```
Component (Result) result
Only the functions and data members that are accessible from your location are displayed
Hint | object returned by the http,ftp,query and mail services
Properties
result | Query
         Template:/Users/markdrew/Dropbox/Railo Team/Book Progress/railo-server-for-demos/webroot/WEB-
         INF/railo/components/org/railo/cfml/Base.cfc
         Execution Time (ms):1
         Recordcount:1
         Cached:No
         SQL:
         SELECT * FROM Users WHERE id =
         '1'
           | id | username
         1 | 1  | user1
prefix | Struct
         cached        | boolean | false
         COLUMNLIST    | string  | id,username
         executionTime | number  | 1
         RECORDCOUNT   | number  | 1
         SQL           | string  | SELECT * FROM Users WHERE id = ?
         sqlparameters | Array
                         1 | string | 1
```

6. The resulting object contains two variables, the result (what comes back from the database) and the prefix (which has information about the query itself).

7. To get the actual query result (so that we can loop it), we add the following code:

```
results = myResult.getResult();
```

What just happened?

By using the built-in `Query` component, we are able to abstract a lot of the semantic noise of using scripted tags. We can get all the information that is required from a query, including the results returned from a database as well as any other variables such as the actual SQL run and the columns that are returned, all in a nice result component.

The HTTP built-in component

Similar to the `Query` component, there are times when you have complex HTTP calls, as we showed in our example, where we obtained the content from the website `http://www.getrailo.org`. HTTP calls to other websites, especially when using REST services, can get a lot more complicated and require a number of parameters. Of course, we could do this using scripted tags, but as you have figured out by now, there is a nice alternative with the built-in components that allows you to do this much easily.

Let's look at how we can make the same code to get the content from an external site, but this time, using a scripted component.

Time for action – getting the content of a website via the HTTP component

1. To start with, let's create a template under `<Railo Install Directory>/webroot/Chapter_6/` named `httpscript.cfm`.

2. In this template, let's add the following code:

```
<cfscript>
  myHTTP = new HTTP(url="http://www.getrailo.org", method="GET");
  myResult =  myHTTP.send();
  dump(myResult.getPrefix().filecontent);
</cfscript>
```

3. When you run this code, you will get a dump of the HTML that is outputted by the website `http://www.getrailo.org`.

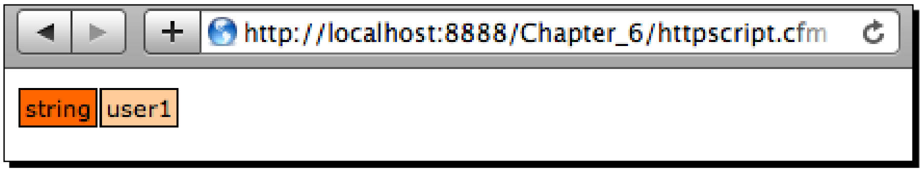
4. But how about passing parameters? Let's try to get the output of a script we created in `queryscript.cfm`. Let's change the script in the `queryscript.cfm` file a bit so that it just returns a username:

```
<cfscript>
  myQuery = new Query(datasource="railobook");
  myQuery.setSQL("SELECT * FROM Users WHERE id = :id");
  myQuery.addParam(name="id", type="cf_sql_numeric", value=url.
id);
  myResult = myQuery.execute();
  results = myResult.getResult();
  WriteOutput(results.username);
</cfscript>
```

5. In the previous code, we changed the final output to just display the username we got from the database. We can now go and call this page from our `httpscript.cfm` if we change it as follows:

```
<cfscript>
  myHTTP = new HTTP(url="http://localhost:8888/Chapter_6/
queryscript.cfm", method="GET");
  myHTTP.addParam(type="URL", name="id", value="1");
  myResult =  myHTTP.send();
  dump(myResult.getPrefix().filecontent);
</cfscript>
```

6. When we load the `httpscript.cfm` in the browser, it will now make another call to our `queryscript.cfm` template and return the following, which is displayed as shown in the following image:



```
string user1
```

What just happened?

With slight modifications, you can use all of the functionality available to you from the tags, via either scripted tags, or even better, by built-in components available to you.

Have a go hero

Can you think of other uses for the `HTTP` component? Why not give it a try to submit forms and get the results from your own scripts?

Railo Server includes other built-in components that you can call in a similar way to the `HTTP` and `Query` components. These are the `Feed`, `Mail`, and `FTP` components. Why not try them out and see what you can do with them?

Summary

In this chapter, we have seen the power of CFScript and how it compares to the tag-based CFML syntax. We have also seen:

- How to use various loops from within CFScript
- How to use CFScript to create your components
- How to use CFML tags inside CFScript
- How to use the in-built components that Railo Server provides

This has been a fun chapter with a lot of coding! In the next chapter, we will be having even more fun as we'll look at the functionality available in Railo Server to create AJAX-powered sites and to convert and display videos.

7
Multimedia and AJAX

In this chapter, we are going to have a look at some of the other functionality that Railo Server provides outside of the normal web application development tooling. In this chapter, we are going to look at:

- Converting and displaying video with `<cfvideo>` and `<cfvideoplayer>` tags
- Adding communication between the browser and Railo Server using the AJAX functionality

Let's start off with checking out some videos.

Video

Video has taken off massively on the Web, and it is often a requirement for a site to be able to use some kind of video functionality. Whether it is for displaying it or converting it, it can be a tough option on how to do it as a developer. Luckily, this is where Railo Server comes to the rescue.

Built into Railo Server is the capability of easily displaying videos on your site. Railo Server does this by enabling a player to display videos by embedding a Flash video player into the HTML of your pages and displaying videos that are encoded using the Flash Video format. You have control over what to display and how it will be displayed, as well as over links, thumbnails, and other options.

The video we are going to use in this chapter is a trailer to the movie Big Buck Bunny, which is the product of the Peach open movie project and is licensed under the Creative Commons Attribution license. This means that we are allowed to use it as our example without licensing issues. If you would like to get a copy of the trailer or the whole movie, you can see it at `http://www.bigbuckbunny.org/`.

Displaying video

Let's see if we can display a trailer video on our own site too:

Time for action – displaying a video player

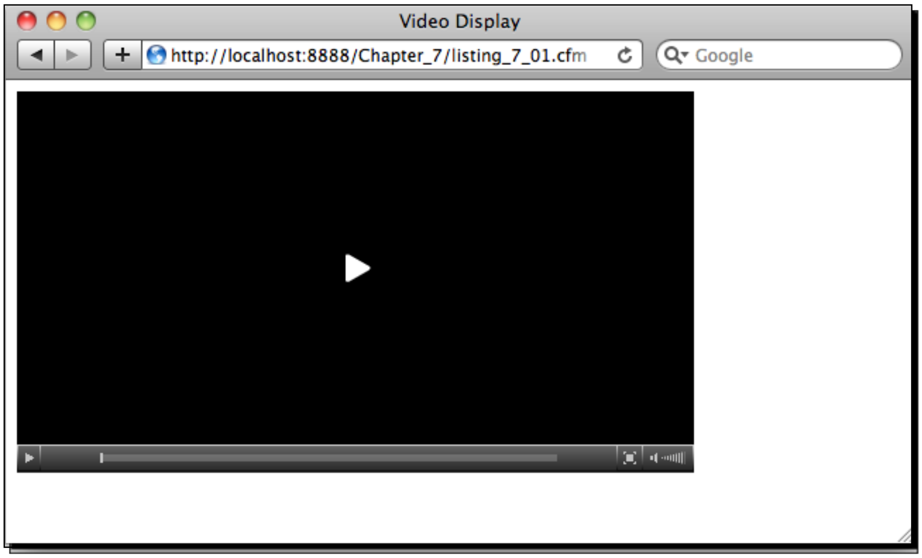
1. Under our `<Railo Install Directory>/webroot/` folder, let's create a `Chapter_7` folder to put all our code samples in.

2. Download and copy the file `trailer.flv` from the code samples, which you can download from the Packt Publishing website. This is the video of Big Buck Bunny that we are going to display.

3. Let's create a template named `listing_7_01.cfm` and add this code, it's the basic outline of a page with the tag `<cfvideoplayer>`:

```
<!DOCTYPE html>
<html>
  <head><title>Video Display</title></head>
<body>
<cfvideoplayer video="trailer.flv" width="480" height="270">
</body>
</html>
```

4. If we now head to `http://localhost:8888/Chapter_7/listing_7_01.cfm`, we will be able to see the video player with our trailer in it. It has the controls that are needed, including play, full screen, and volume control.



5. The attributes of the `<cfvideoplayer>` tag are quite easy; we have the video that we want to display as well as the `width` and the `height` to display this video.

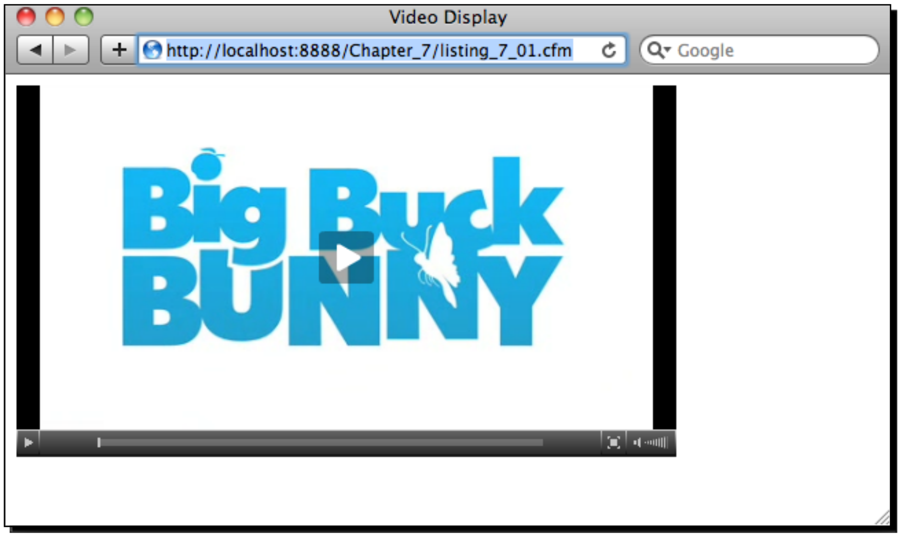
6. At the moment, our video is kind of boring because until we click on the play button, there is no image; just a black square. Let's add a preview image.

7. Copy the `trailer.jpg` file from the code samples' directory you obtained from the Packt Publishing website into the `Chapter_7` directory.

8. Edit the code in `listing_7_01.cfm` and add the `preview="trailer.jpg"` attribute to the tag, so our code should look like this now:

```
<!DOCTYPE html>
<html>
  <head><title>Video Display</title></head>
<body>
<cfvideoplayer video="trailer.flv" width="480" height="270"
thumbnails="true" preview="trailer.jpg">
</body>
</html>
```

9. When you now reload the page by going to `http://localhost:8888/Chapter_7/listing_7_01.cfm`, you will see a nice background to the video:



http://localhost:8888/Chapter_7/listing_7_01.cfm

What just happened?

By adding the `<cfvideoplayer>` tag, we can display a video in the page as long as the video is in the FLV video format. The `<videoplayer>` tag accepts the `preview` attribute that allows us to assign a poster frame to it.

Converting a video

Before we continue exploring all the options available in the `<cfvideoplayer>` tag, we really should explore the video conversion capabilities that Railo Server can offer. These capabilities are not actually part of the base install; they are provided in the form of a Server Extension. Server Extensions provide a way to add functionality to Railo Server (you will see how to create your own in Chapter 9).

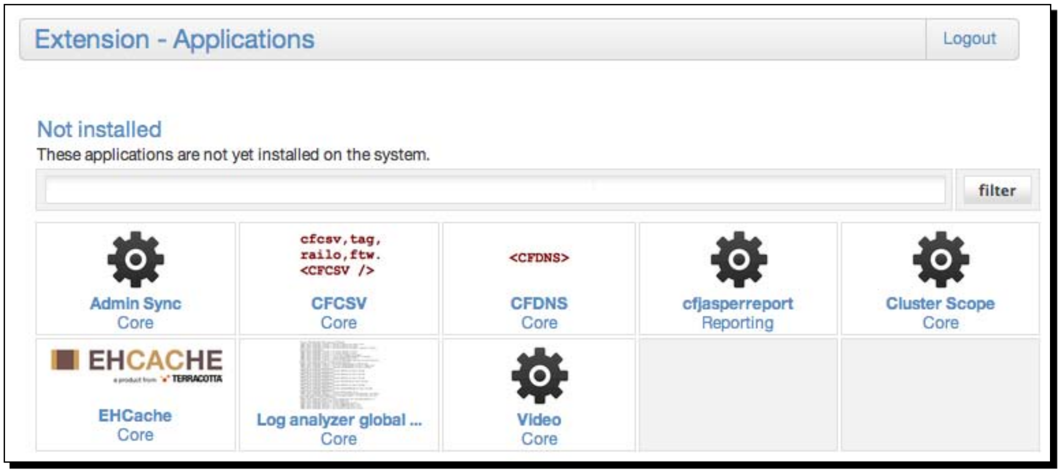
The CFVideo Extension allows you to convert video, extract single frames, and get information on a variety of video formats out there. Rather than just talking about it, let's go and get it installed.

Time for action – installing the Video Extension

So far, we have looked at the Railo Web Administrator, but because the Video Extension is installed server wide, we need to go to the Railo Server Administrator to download the extension.

1. In your browser, go to `http://localhost:8888/railo-context/admin/server.cfm` and either log in or enter your new password.

2. Under the **Extension** section on the left-hand side, click on **Applications**. This will take you to the **Applications** page where you can see the available extensions for Railo Server.

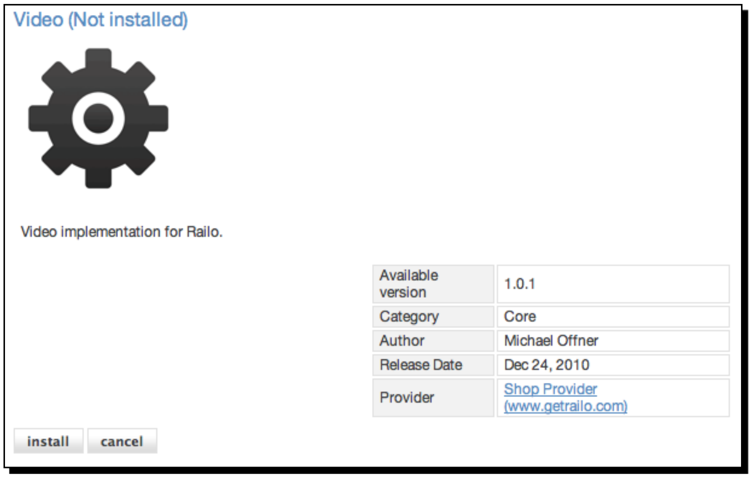


Extension - Applications

Not installed

These applications are not yet installed on the system.

3. Click on the **Video Core** extension, and it will take you to the information page of the Video Extension.

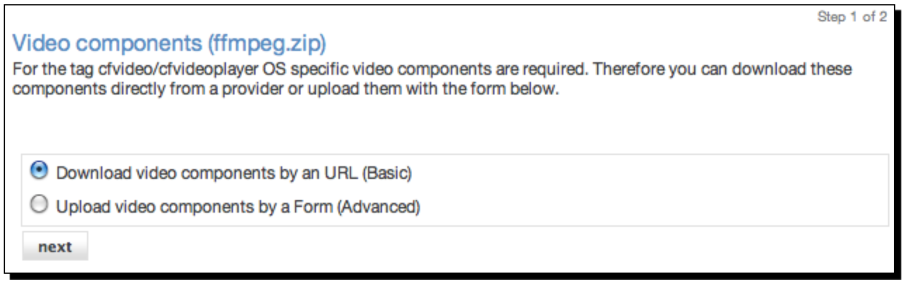


Video (Not installed)

Video implementation for Railo.

Available version	1.0.1
Category	Core
Author	Michael Offner
Release Date	Dec 24, 2010
Provider	Shop Provider (www.getrailo.com)

4. Click on the **Install** button, and it will take you to the page that allows you to install the video components. Under the hood, the `<cfvideo>` tag uses the FFmpeg library. This form allows you to install your own version or get a pre-built version for your operating system.



Step 1 of 2

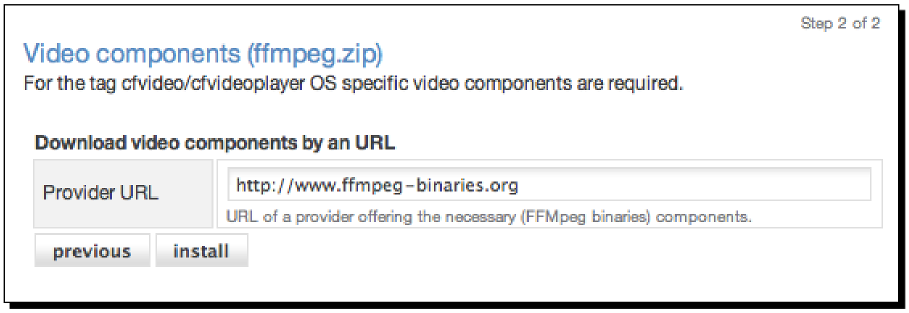
Video components (ffmpeg.zip)

For the tag cfvideo/cfvideoplayer OS specific video components are required. Therefore you can download these components directly from a provider or upload them with the form below.

● Download video components by an URL (Basic)

○ Upload video components by a Form (Advanced)

5. Leave the radio button selected on the **Download video components by an URL** item and click on **next**. This takes you to the URL selection screen, which you can leave as is, and click on **install**.



Step 2 of 2

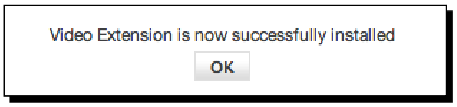
Video components (ffmpeg.zip)

For the tag cfvideo/cfvideoplayer OS specific video components are required.

Download video components by an URL

Provider URL	http://www.ffmpeg-binaries.org URL of a provider offering the necessary (FFMpeg binaries) components.

6. This will start the install process, and after a few seconds, you should have a notice saying that the Video Extension has been successfully installed.



Video Extension is now successfully installed

What just happened?

There are a lot of functionalities that don't need to be deployed with the Railo Server itself. We are able to extend the Railo Server using Applications that are produced either by Railo technologies or other content providers in many ways. In the previous example, we were able to add a new tag, namely, the `<cfvideo>` tag, which we can use to convert a video.

Now that we have the `<cfvideo>` tag installed, we can get on with playing with the conversion of the video. Let's get to it!

Time for action – creating clips for our video player

In order to display the video, we used a file named `trailer.flv`. This file was actually generated by the `<cfvideo>` tag itself. Let's do that again, but from scratch. Don't worry, it won't take too long.

1. First off, let's get the original video. Copy the file named `bbb_trailer_iphone.m4v` from the code samples into the `Chapter_7` folder.
2. Delete the current `trailer.flv` file that we have in the code samples; we will create a new one from the `bbb_trailer_iphone.m4v` file.
3. Now, let's create a file that will do our conversion for us. Create a template named `listing_7_02.cfm` and put in the following code:

```
<cfvideo action="convert"
source="bbb_trailer_iphone.m4v"
destination="trailer.flv">
```

4. We can now run `http://localhost:8888/Chapter_7/listing_7_02.cfm`, and after a few seconds, a new `trailer.flv` will be created.
5. Before we go on, we should add some more code. Because videos take a long time to convert, we need to add some code that will tell Railo Server to allow this template to run for longer than the default time.

6. At the top of `listing_7_02.cfm`, add a `<cfsetting>` tag so that your code looks as follows:

```
<cfsetting requesttimeout="600">
<cfvideo action="convert" source="bbb_trailer_iphone.m4v"
destination="trailer.flv">
```

What just happened?

Using the `<cfvideo>` tag, we are able to convert to a number of formats. The `<cfvideo>` tag is intelligent enough to realize what you want to convert the video to and then do the conversion for us.

If we now go back to our player that we had in the `listing_7_01.cfm` template and run it by going to `http://localhost:8888/Chapter_7/listing_7_01.cfm`, we will see the video that we just converted.

Let's create another clip from the trailer and create a couple of poster frames for it so that we can display a playlist next to our standard player.

Time for action – creating poster frames and clips

We already saw how to convert one video. Let's create another video, but this time we are going to define which segment of the video we want to convert, rather than converting the whole video.

1. In the `listing_7_02.cfm` template, add the following lines of code so that the template code now looks like this:

```
<cfsetting requesttimeout="600">
<cfvideo action="convert" source="bbb_trailer_iphone.m4v"
destination="trailer.flv">
<cfvideo action="convert" source="bbb_trailer_iphone.m4v"
destination="clip.flv" start="19s" max="3s">
```

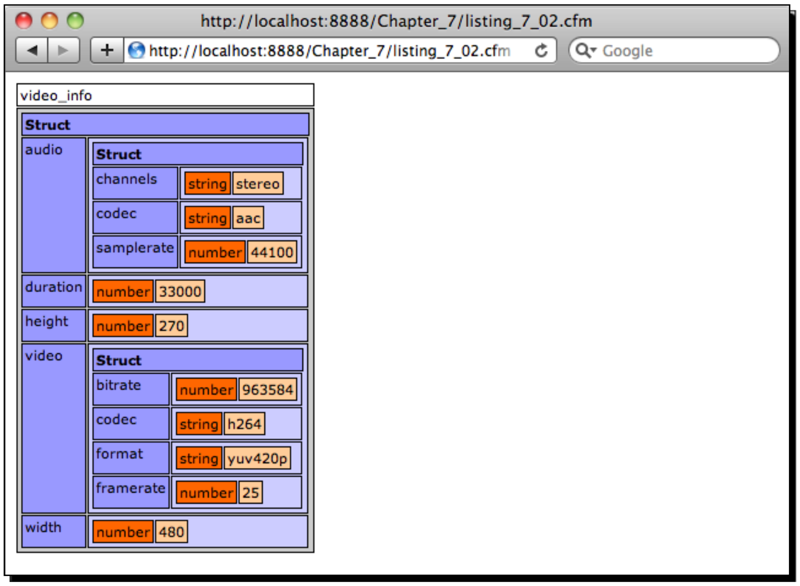
2. When we run the code by going to `http://localhost:8888/Chapter_7/listing_7_02.cfm` and look in the folder, you will see we have a `trailer.flv` file and a `clip.flv` file. The clip was created by stating when we wanted the conversion to start, and the number of seconds we wanted to convert (the `start="19s"` and `max="3s"` attributes).

3. Awesome! We have now created our clips.

4. Let's get some info from the video to create our poster images, we can do this by selecting another action to perform with the `<cfvideo>` tag; this time we can add the following:

```
<cfsetting requesttimeout="600">
<cfvideo action="convert" source="bbb_trailer_iphone.m4v"
destination="trailer.flv">
<cfvideo action="convert" source="bbb_trailer_iphone.m4v"
destination="clip.flv" start="19s" max="3s">
<cfvideo action="info" source="bbb_trailer_iphone.m4v"
result="video_info">
<cfdump eval=video_info>
```

5. When we run this code, we can now see that the `<cfvideo action="info" source="bbb_trailer_iphone.m4v" result="video_info">` returns a structure with information about the video. What we are interested in are the width and height variables.


http://localhost:8888/Chapter_7/listing_7_02.cfm
http://localhost:8888/Chapter_7/listing_7_02.cfm
Google

```
video_info
Struct
audio      Struct
           channels    string stereo
           codec       string aac
           samplerate  number 44100
duration   number 33000
height     number 270
video      Struct
           bitrate     number 963584
           codec       string h264
           format      string yuv420p
           framerate   number 25
width      number 480
```

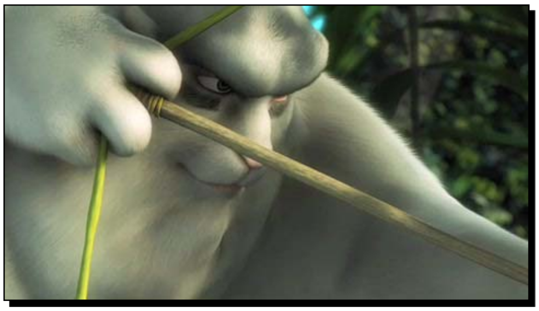
6. Let's use these variables to create the image files for the video by adding another `<cfvideo>` tag action, namely, the `cutImage` action. Let's add the code to our template:

```
<cfsetting requesttimeout="600">
<cfvideo action="convert" source="bbb_trailer_iphone.m4v"
destination="trailer.flv">
<cfvideo action="convert" source="bbb_trailer_iphone.m4v"
destination="clip.flv" start="19s" max="3s">
```

```
    <cfvideo action="info" source="bbb_trailer_iphone.m4v"
    result="video_info">
    <cfdump eval=video_info>

    <cfvideo action="cutimage"
    source="bbb_trailer_iphone.m4v" destination="trailer.jpg"
start="28s" width="#video_info.width#" height="#video_info.height#">
    <cfvideo action="cutimage"
    source="bbb_trailer_iphone.m4v" destination="clip.jpg"
start="19.5s" width="#video_info.width#" height="#video_info.height#">
```

7. When you run the code, you will see the dump of the `video_info` variable. If you go into the folder, you should now see a couple of new files, `trailer.jpg` and `clip.jpg`, which will be used in our video player! If you look at these files, they should look like the following images:

Big Buck
BUNNY



What just happened?

Now that we have installed the `Video` extension to Railo Server, we can use the various functions of the `<cfvideo>` tag. We converted a video to FLV for use in our `<cfvideoplayer>` tag, we obtained information about the video using the `action="info"` attribute of the tag, and finally, we created a poster image by using the `action="cutImage"` and defining at which point we wanted to cut the image from the video.

Now that we have done all the preparatory work, let's go and add a playlist to our `<cfvideoplayer>`.

Time for action – adding a playlist to <cfvideoplayer>

Let's go and add some movies to our playlist:

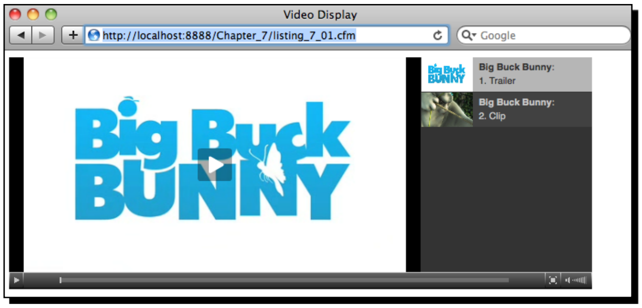
1. Open up the template we were using to display our `<cfvideoplayer>` `listing_7_01.cfm` and edit the code. We are going to replace the video player we have there with the following code:

```
<!DOCTYPE html>
<html>
  <head><title>Video Display</title></head>
<body>
  <cfvideoplayer playlist="right"  playlistsize="200"
playlistthumbnails="true" width="480"  height="270">
    <cfvideoplayerparam video="trailer.flv" preview="trailer.jpg"
author="Big Buck Bunny" title="1. Trailer">
    <cfvideoplayerparam video="clip.flv" preview="clip.jpg"
author="Big Buck Bunny" title="2. Clip">
  </cfvideoplayer>
</body>
</html>
```

2. As you can see, we have a new set of attributes. We have a `playlist="right"` which defines where we want to display the playlist, the `playlistsize="200"` which defines the width of the playlist, and `playlisthumbnails` which defines if we are going to show thumbnails in our playlist.

3. In the body content of the `<cfvideoplayer>` tag, we have some new tags, namely, `<cfvideoplayerparam>`tags. These tags are used to define the videos we are going to have in the video player. With the attribute `video="clip.flv"`, we define which videos we are going to show, and with the `preview="clip.jpg"` attribute, we define which is the preview or poster image we are going to display. The `author="Big Buck Bunny"` sets an author to display in the playlist, and finally, we can add a `title` attribute to display it.

4. Let's have a look at our player now. Head to `http://localhost:8888/Chapter_7/listing_7_01.cfm` to view it.



What just happened?

Once we have prepped our resources, we can use the <cfvideoplayer> tag to display a playlist of videos, rather than just one video. Using the <cfvideoplayerparam> tag, we can add new videos and set a number of variables such as the author and the title.

Hopefully, this section has encouraged you to use the video feature a bit more. There are many more options with all these tags, but this has just been a quick tour through some of their functionality.

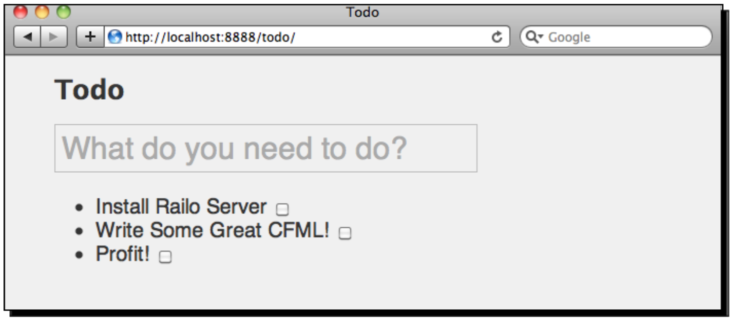
You can see more attributes of the <cfvideo> tag in the Railo Wiki: http://wiki.getrailo.org/wiki/TAG:CFVIDEO.

AJAX functionality within the Railo server

As we have just seen, using and manipulating a video is pretty easy with Railo Server. But that is not all. Railo Server also allows you to add AJAX (Asynchronous JavaScript and XML) functionality to your web applications with ease.

AJAX allows you to build dynamic frontends that don't need to refresh the page to show results from the server. There are many JavaScript libraries out there that make communicating with the server easy, but as we shall see, Railo Server makes it even easier.

For this section, we are going to build a simple application to store our tasks. For simplicity, we are just going to store our tasks in the session scope, but, if you want, you can save them to the database using the ORM capabilities.



Todo

What do you need to do?

- Install Railo Server □
- Write Some Great CFML! □
- Profit! □

Here's what we are going to build from scratch. It's basically a form that you can enter a task. The entered tasks are listed as shown in the previous screenshot, and you can delete a task by just clicking on the checkbox associated with each task. Simple enough functionality, but to make it more interactive, we are going to be using the power of Railo Server's AJAX functionality to build this application.

Let's get started!

Time for action – setting up the application and services

Before we look at the AJAX functionality, let's set up our Railo Application and create the server-side service that will be used to store our tasks:

1. Create a directory named `todo` in your `<Railo Install Directory>/webroot` directory; this is where we are going to hold the application.

2. Now let's create the `Application.cfc` for this application, so add a template called `Application.cfc` and add the following code:

```
component {
  this.name = "TodoList";
  this.sessionmanagement = true;

  function onSessionStart(){
    session.tasks = [];
  }
}
```

3. Next, we are going to create our `TaskService`. Let's create a template named `TaskService.cfc` in the `todo` folder and add the following code:

```
component{

  remote function addTodo(String taskname){
    if(Len(arguments.taskname)){
     ArrayAppend(SESSION.tasks, {name:arguments.taskname,
     addedat=Now()});
    }
    return SESSION.tasks;;
  }

  remote function removeTodo(Numeric id){
    ArrayDeleteAt(SESSION.tasks, id);
    return SESSION.tasks;
  }
}
```

4. In the previous code, we have created a component with two functions, the `addTodo()` and the `removeTodo()`. We notice that we have a `remote` keyword before the function. That tells the component that you can call these functions remotely, either as web services or via JavaScript, and they will return **JSON** (**JavaScript Object Notation**) objects. The `addTodo` function takes a task name as a variable, checks whether there is anything in it (in other words, it isn't a blank string), and if there is something, it appends a new entry in the array that is made up of a structure with the name of the task and when we added it. The remove function is pretty simple, it uses the `ArrayDeleteAt()` function to delete the entry in the array with the same position. We shall see how this works a bit later.

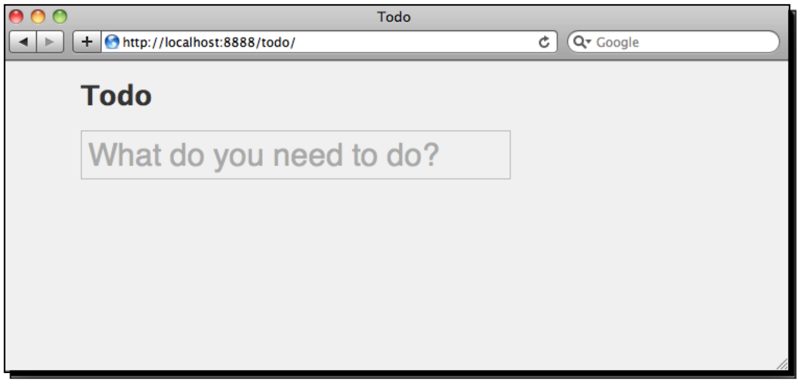
5. Now that we have added these two functions, we can start building our page.

6. Let's copy the stylesheet named `main.css` from the `code_samples` directory into our `todo` directory, so that we have a nice style to get going with.

7. Next, let's create a template named `index.cfm`. This is where most of the action is going to happen, as you add the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="main.css" type="text/css">
  <title>Todo</title>
</head>
<body>
  <div id="page">
    <h1 id="todo">Todo</h1>
    <input type="text" name="taskname" id="taskname"
placeholder="What do you need to do?">
  </div>
</body>
</html>
```

8. The previous code is simply the outline of the page, so we have everything set up and it should look something like the image below. There is no functionality as yet, but it's a good start!



What just happened?

The code in `Application.cfc` gives our application the name `TodoList` and enables the `SESSION` scope for use by setting `this.sessionmanagement=true`. So that we know we have a task array ready to use, we make sure that when the session starts (with the function `onSessionStart(){}`) we have an empty array ready for us to fill.

We then created the TaskService.cfc so that we can add and remove tasks from the SESSION scope, and finally we have put the index.cfm main file and given it a nice stylesheet so it doesn't look too bare. The Next we will be to add new tasks.

Time for action – binding the input to the component

Because we want to be able to enter items in the main text field, let's create a binding form that form-inputs to our TaskService.cfc.

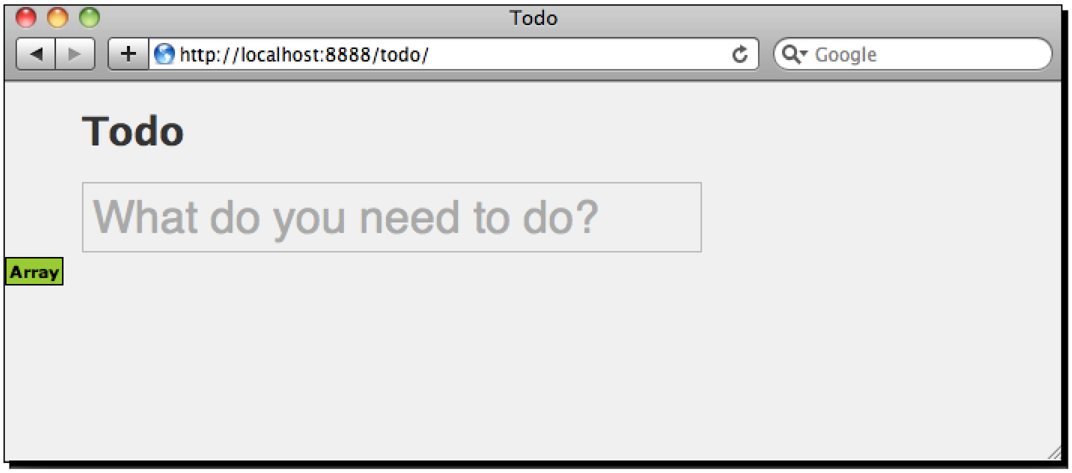
1. Add the following code to index.cfm; this will bind the text field to the TaskService.cfc:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="main.css" type="text/css">
  <title>Todo</title>
  <cfajaxproxy bind="cfc:todo.TaskService.addTodo({taskname})"
         onSuccess="displayTodos"
         onError="onError"/>
  <script type="text/javascript" charset="utf-8">
      onError = function(code,message){
        alert(code + ' - ' + message);
      }
      displayTodos =   function (data){
        document.getElementById('taskname').value = "";
      }
  </script>
</head>
<body>
  <div id="page">
    <h1 id="todo">Todo</h1>
    <input type="text" name="taskname" id="taskname" 
placeholder="What do you need to do?">
  </div>

  <cfdump var="#SESSION.TASKS#">
</body>
</html>
```

2. In the previous highlighted code, we add the `<cfajaxproxy>` tag. This tag allows you to bind to a component defined by the `bind="cfc:todo.TaskService.addTodo({taskname})"` attribute. There is a lot going on in here, but in essence, we are binding to the `addTodo()` function in the `TaskService.cfc` template. We are then referencing the `taskname` field (notice the name of the `<input type="text" name="taskname" ... >` field) by name. Any changes that happen to that field, will then call this function, without us having to do anything about it.

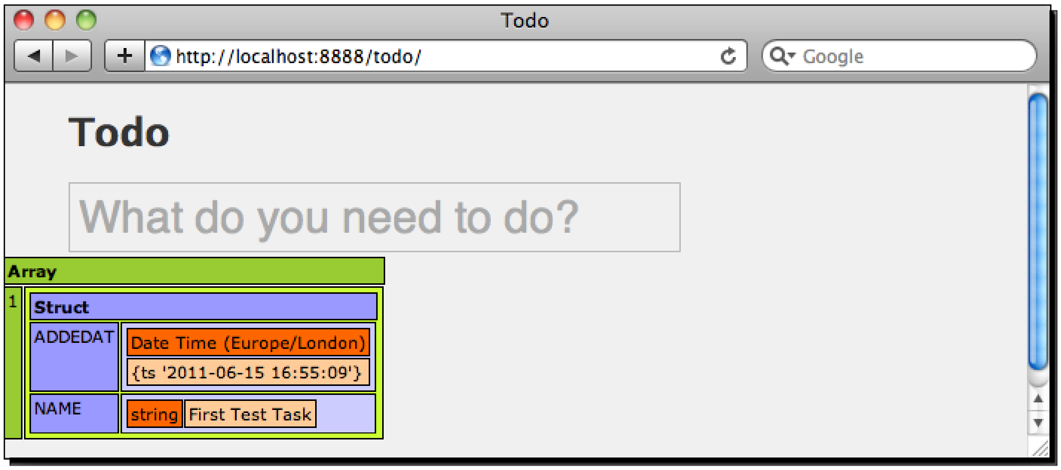
   You also notice we are adding a couple of JavaScript functions; the first one is `onError`, which is triggered if there are any errors with the call. The second is `onSuccess="displayTodos"`, which gets called when the call is successful. The final bit of code is just for debugging, so now if you add something in the text field, it will update the session without reloading, but we won't see anything happen. To see any changes, we need to reload the page so that we can see the array of tasks that is displayed using `<cfdump var="#SESSION.TASKS#">`

   Let's give it a go to `http://localhost:8888/todo/` and you will see an empty array of tasks:



3. Enter a task and press the return key; nothing will happen apart from the entry field going back to blank

4. If you now reload the page, you will see that we have successfully added another item to the session:



What just happened?

Using the <cfajaxproxy>, we bounded events that happen on the taskname field to the addTodo() function on the TaskService.cfc component. A lot is happening behind the scenes that we don't have to worry about because Railo Server is taking care of all the wiring up for us.

Because we are saving the tasks to the SESSION scope, we needed to put some debug code to display; but let's change that now and create a way to display our items.

Time for action – displaying the tasks

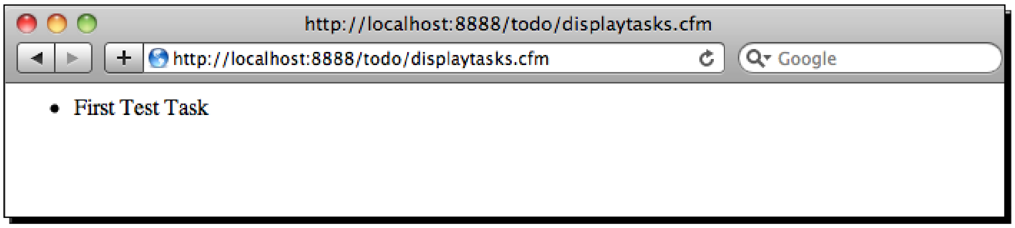
Because we have to reload the page to show our tasks. Let's do this automatically? This is really simple. But First, let's create a page that displays the tasks:

1. Create a template named displayTasks.cfm and save it in the todo folder.
2. In the displayTasks.cfm template, add the following simple loop code:

```
<ul id="taskList">
  <cfloop array="#SESSION.tasks#" index="task">
  <li><cfoutput>#task.NAME#</cfoutput></li>
  </cfloop>
</ul>
```

3. The previous code simply loops through the items in the SESSION.tasks array and displays them in an unordered list.

4. If we now go to `http://localhost:8888/todo/displaytasks.cfm`, we can see the output from the session:


http://localhost:8888/todo/displaytasks.cfm
http://localhost:8888/todo/displaytasks.cfm
Google

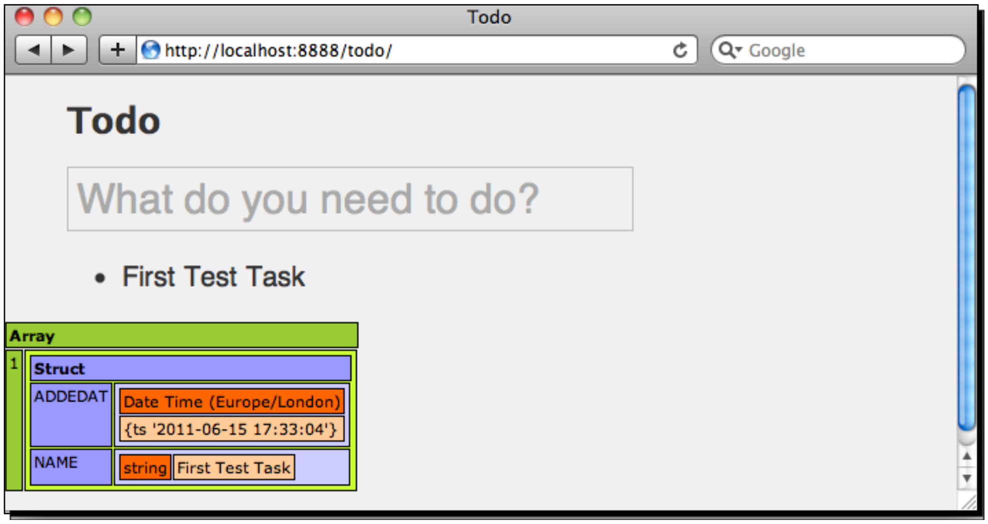
- First Test Task

5. Now that we have something to display, we can integrate it into the main `index.cfm` template. Let's add another AJAX tag to display the contents of `displayTask.cfm`; change the main part of your code to add the `<cfdiv>` tag:

```
<div id="page">
  <h1 id="todo">Todo</h1>
  <input type="text" name="taskname" id="taskname" 
placeholder="What do you need to do?">
  <cfdiv id="displayTodos" bind="url:displayTasks.cfm"></cfdiv>
</div>
```

6. We can see the `<cfdiv>` tag in action. We give it an `id` so that we can refer to it by name, and then we use the `bind="url:displayTasks.cfm"` attribute to bind the contents of the div to our `displayTasks.cfm` template.

7. If you reload the page by going to `http://localhost:8888/todo/`, you will now see your tasks displayed.



Todo

What do you need to do?

- First Test Task

Array			
1	Struct		
	ADDEDAT	Date Time (Europe/London)	
		{ts '2011-06-15 17:33:04'}	
	NAME	string	First Test Task

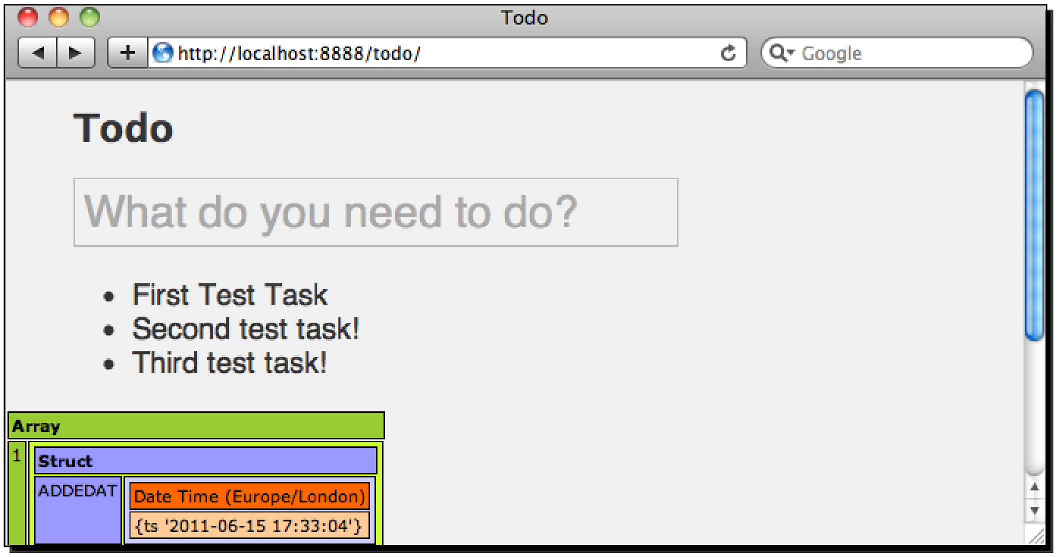
8. We are not finished yet. We still have to reload the whole page to get them to display. How about we add functionality so that once we add a task, the `<cfdiv>` tag automatically refreshes? This is easy since we already have an `onSuccess` function we are triggering.

9. Change the `<script>` block so that it has the following code:

```
<script type="text/javascript" charset="utf-8">
    onError = function(code,message){
      alert(code + ' - ' + message);
    }
    displayTodos =   function (data){
      document.getElementById('taskname').value = "";
      Railo.Ajax.refresh('displayTodos');
    }
</script>
```

10. In the previous code, we have added one of the built-in JavaScript objects that Railo Server has placed in our page. By calling `Railo.Ajax.refresh('displayTodos')`, we are telling the Railo Ajax object to refresh our `<cfdiv>` since we are referring to it by its ID.

11. Refresh the page and you can now enter another item. It will be displayed immediately without a page refresh between submissions.



Todo

What do you need to do?

- First Test Task
- Second test task!
- Third test task!

```
Array
1 Struct
  ADDEDAT  Date Time (Europe/London)
           {ts '2011-06-15 17:33:04'}
```

What just happened?

By using the `<cfdiv>` tag, we were able to bind it to a URL, namely, that of our `displayTasks.cfm` page that lists our tasks. When we use any Railo Server AJAX tags, we automatically have a Railo JavaScript object available that allows us to interact with various parts of the page, including other Railo AJAX tags and, for example, be able to reload the `<cfdiv>`.

So far so good! But what about when we have completed the task already? Why don't we go ahead and add the ability to delete a task?

Time for action – deleting a task

If you remember, in the `TaskService.cfc` template, we have a function called `removeTodo()` that takes the position of the task in our array and deletes it. Let's connect a JavaScript function to this server-side function:

1. In the `index.cfm` template, let's add another `<cfajaxproxy>` tag. This time it will bind the `TaskService.cfc` to a JavaScript variable:

```
<head>
  <link rel="stylesheet" href="main.css" type="text/css">
  <title>Todo</title>
  <cfajaxproxy bind="cfc:todo.TaskService.addTodo({taskname})"
         onSuccess="displayTodos"
         onError="onError"/>
  <cfajaxproxy cfc="todo.TaskService"" jsclassname="TaskService">
```

2. In the previous code, we have used the `<cfajaxproxy>` tag again, but this time we are using the `cfc="todo.TaskService"` attribute to say which component we want to bind to a new JavaScript object. We then use the `jsclassname="TaskService"` attribute to set its name.

3. Now that we have bound it to a new object, let's add the JavaScript function to delete an item:

```
<script type="text/javascript" charset="utf-8">
    onError = function(code,message){
      alert(code + ' - ' + message);
    }
    displayTodos =  function (data){
      document.getElementById('taskname').value = "";
      Railo.Ajax.refresh('displayTodos');
    }
    markDone = function(item){
      var Todo = new TaskService();
```

```
        Todo.removeTodo(item);
        Railo.Ajax.refresh('displayTodos');
      }
</script>
```

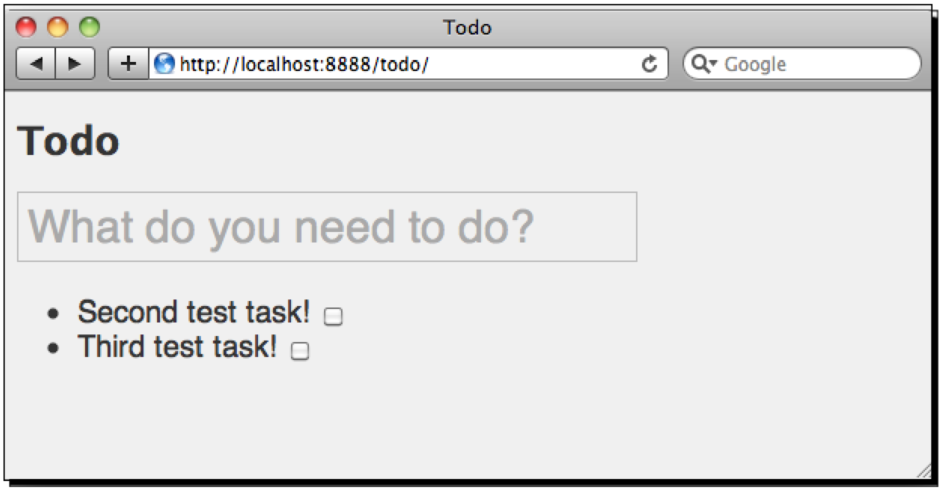
4. In the previous code snippet, we added the `markDone` JavaScript function. When this function is called, it will pass an `itemid` which we will call from each of the items in the list. Then, it will create an instance of the `TaskService` JavaScript object we have defined with the `<cfajaxproxy>` tag. We can then call the `removeTodo()` function and pass it the position of the item (`item`) to delete it. Once we have done that, we can re-run the `Railo.Ajax.refresh()` function to re-display our tasks.

5. Now that we have all that in place, we need to trigger this function from each of the displayed tasks, so let's edit our `displayTasks.cfm` template:

```
<ul id="taskList">
  <cfset counter = 1>
  <cfloop array="#SESSION.tasks#" index="task">
  <li><cfoutput>#task.NAME# <input type="checkbox"
value="#counter#" onClick="markDone(#counter#)"></cfoutput></li>
  <cfset counter++>
  </cfloop>
</ul>
```

6. In the previous code, we are adding a checkbox after the name of the task. So that we tick each task as done, we add an `onClick=""` JavaScript attribute to the `<input>` box. This calls the `markDone()` function we just defined. Since we need to know which item we want to delete, we create a counter. After each loop we increase it by one with the `<cfset counter++>` call, and then use the counter variable in our method call to `markDone(#counter#)`.

7. When we now reload the page, we can see each of the items, and when we click on the checkbox, the code will call the `markDone` function, which in turn will call the `removeTodo()` function in our `TaskService.cfc` component. Finally, we use the Railo JavaScript object to refresh the `<cfdiv>` with our tasks.



Todo

What do you need to do?

- Second test task! □
- Third test task! □

What just happened?

By using the `<cfajaxproxy>`, we are able to bind a proxy to a component to a JavaScript object. This JavaScript object can then be used directly from JavaScript to call remote methods on a component. It's really just that simple! With this method, we were able to assign an event handler to the task checkbox, which, when called, called a JavaScript function to call our component.

Summary

In this chapter, we covered a couple of the multimedia and AJAX capabilities of Railo Server:

- We used the `<cfvideoplayer>` tag to display videos
- We also used the `<cfvideoplayer>` tag to control which preview image we want to assign to a video and even add other videos in a playlist
- Using the Extension Store, we added the Video Extension so that we had access to the `<cfvideo>` tag
- Using the `<cfvideo>` tag, we converted different video formats so that they can be displayed in our web page using the `<cfvideoplayer>` tag
- We also created clips and thumbnails from our origin video using the `<cfvideo>` tag

In the second part of the chapter, we looked at Railo Server's AJAX functionality. We looked at:

- Using the `onSessionStart()` method in the `Application.cfc` to set up a `SESSION` variable to store our tasks
- We created a `TaskService.cfc` to add and remove tasks from the `SESSION.tasks` variable
- We used the `<cfajaxproxy bind="cfc:todo.TaskService.addTodo({taskname})" >` to bind changes to an input field to a method in our `TaskService.cfc` component
- We used the `<cfdiv>` tag to include an external page within our main page dynamically
- We then used the built in Railo AJAX functions to refresh the contents of the `<cfdiv>` tag when we updated the tasks using the `Railo.Ajax.refresh()` function

In the next chapter, we are going to look at Mappings and Resources that are available to you when using Railo Server. This will show you how you can store items in memory, have consistent paths that you can change to suit your needs in the Railo Server administrator, and how to use Amazon's Simple Storage Service to your advantage.

Onwards we go!

8
Resources and Mappings

By now, you should have a good handle on developing different applications with Railo Server. In this chapter, we will be going through the resources and mappings that Railo Server is able to use as filesystems to access your files. In short, we will be looking at:

- Accessing files locally and then creating mappings as aliases to files and folders
- Accessing your files from ZIP and TAR files
- Using RAM as a quick location to store files
- Using Amazon's Simple Storage Service to place files in the Cloud

By the end of this chapter, you will see the benefits of using mappings as well as being able to create clustered solutions that access single resources.

Let's get started!

Railo resources

The architecture of Railo Server has been created so that everything is coded to an interface. This means that most parts of the Railo Server can be extended to use different kinds of underlying systems. Let's take the example of datasources. All the datasources in Railo Server comply with an interface that we have defined. This means that if we want to add another type of database, we just need to create a driver that has the same functionality, as defined in the interface, and it will work with Railo Server nicely.

This goes for resources too. So, for example, you can access files in the filesystem with the various `FileXXX()` or `DirectoryXXX()` functions as well as the `<cffile>` and `<cfdirectory>` tags. You might think that there is only one type of filesystem, namely, that of the hard drive installed in the server. However, this is not the case. There are other filesystems out there that we can use, for example, FTP servers. You should be able to list a number of files, read, write those files, and delete, the same way you do on your local filesystem.

Accessing files locally

There are many times when you want to access files locally. For example, you might want to list the images that a user has uploaded, write to log files and even include code that is in a different file. Railo Server provides a number of functions for this; they are usually named `FileXXX()` or `DirectoryXXX()` where `XXX` is the action to be performed. Railo Server also provides these as tags, in the form of the `<cffile aciton="XXX">` and `<cfdirectory action="XXX">`, again, where the `XXX` is the action to be performed.

Let's look at some examples.

Time for action – writing and reading files

Let's say that we want to log some information about what is happening in our application; rather than displaying it to the user. We should store this in a file that we can read either manually or via another interface, for example, in an administration application we might want to build.

Let's create the simplest code for this:

1. First off, let's create a folder to keep our logs. In `<Railo Installation>/webroot/Chapter_8/`, create a folder named `logs`

2. Now, let's write some code to append to a log file inside it (we haven't created any files inside it, but Railo Server will take care of it). Create a file in `<Railo Installation>/webroot/Chapter_8/` named `listing_8_01.cfm` and add the following:

```
<cffile action="append" file="logs/mylog.txt" output="This is the
output to our log file! #Now()#">
```

3. If we now load up the script a few times by going to `http://localhost:8888/Chapter_8/listing_8_01.cfm`, we can populate our log file and you should see something like this if you open up the file in `<Railo Installation>/webroot/Chapter_8/logs/mylog.txt`:

   ```
   This is the output to our log file! {ts '2011-06-11 14:53:01'}
   This is the output to our log file! {ts '2011-06-11 14:53:04'}
   This is the output to our log file! {ts '2011-06-11 14:53:16'}
   This is the output to our log file! {ts '2011-06-11 14:53:29'}
   ```

4. Rather than opening the file, let's modify our script to also list what the log file consists of:

   ```
   <cffile action="append" file="logs/mylog.txt" output="This is the
   output to our log file! #Now()#">
   <cffile action="read" file="logs/mylog.txt" variable="MyLogs">
   <pre>
   <cfoutput>#MyLogs#</cfoutput>
   </pre>
   ```

5. If we now reload the template by going to `http://localhost:8888/Chapter_8/listing_8_1.cfm`, we should now see the contents of the log file! Awesome, huh?

What just happened?

Using some of the simple tags available in Railo Server, we were able to log messages to a file, read that file, and display its contents on screen quite easily.

Looping files

The problem with the preceding code is that it gets all the contents of the file, which is rather annoying. What happens if we want to loop through each line in the file instead? Well, that's what the `<cfloop>` is for.

Time for action – looping through the contents of a file

Let's change our code in the `listing_8_01.cfm` template to loop through the contents of a file; this will allow us to have a bit more control over what we do with the output.

1. Edit `listing_8_01.cfm`, remove the `<cffile action="read">`, and let's use a loop to display each line of the code:

```
<cffile action="append" file="logs/mylog.txt" output="This is the
output to our log file! #Now()#">
<cfloop file="logs/mylog.txt" index="i">
  <cfoutput>#i#</cfoutput><br>
</cfloop>
```

2. If you now run the template by going to `http://localhost:8888/Chapter_8/listing_8_01.cfm`, you will get each line outputted nicely. You could now parse each line.

What just happened?

The `<cfloop>` tag has the `file` attribute which lets us bypass reading the file directly, as the file could be rather large. Using the `<cfloop file="">` tag, we can loop through every line in a very large file and parse it without worrying about the size.

Mappings

The previous examples are fine if the code was always on that server, but what if we wanted to move our code to another server, or even worse, if that new server had a different operating system? Also, what would happen if we wanted to move the location of the logs to another disk (for example, if it was getting too big)?

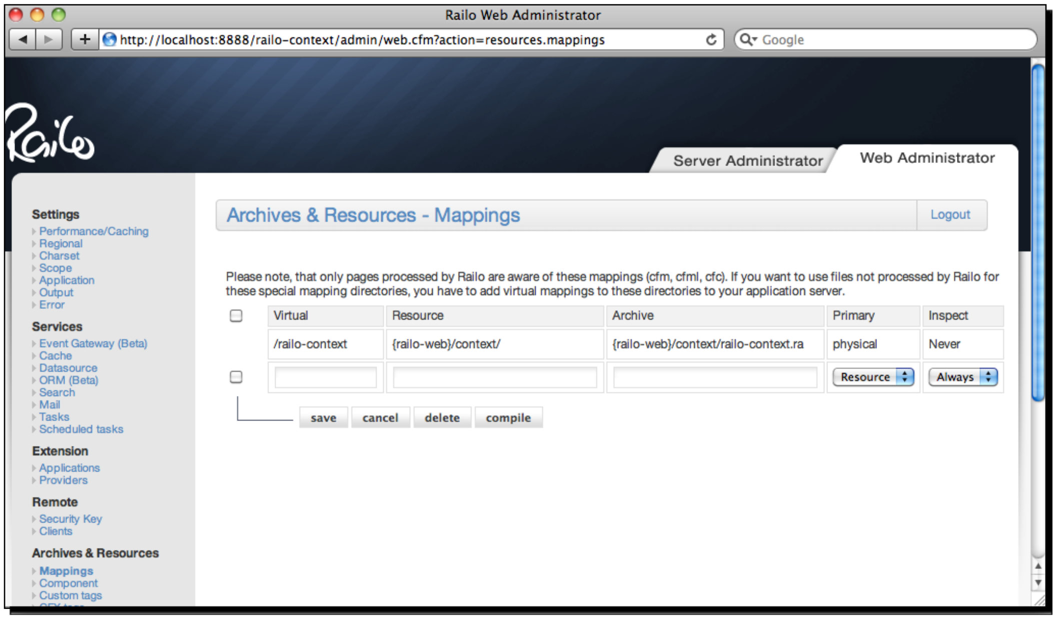
This is where the idea of mappings comes in. Mappings in Railo Server are ways to create a shortcut to a folder on another part of the server. This makes your code more portable.

For example, imagine if we had an application that wrote to a specific file (or read files from a specific directory) such as `C:\MyApplication\MyLogfiles\usercount.txt` and we had that path written all over our code. It would then be a nightmare of searching and replacing throughout our code to change that. With mappings, we can create a link in the Railo Server Administrator and manage the location of this file outside our code. Let's do this for our log file.

Time for action – creating a mapping for the log file

Instead of hardcoding the location of our code, let's create a mapping in the Railo Web Administrator that we are going to use to point to our location.

1. Open up the Railo Web Administrator by browsing to `http://localhost:8888/railo-context/admin/web.cfm` and log in.
2. Click on the **Mappings** link that can be found under the **Archives & Resources** section on the left.
3. Here you will see the **Archives & Resources - Mappings** screen.

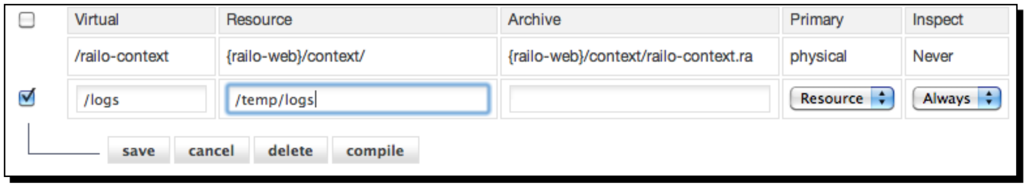


Archives & Resources - Mappings

Please note, that only pages processed by Railo are aware of these mappings (cfm, cfml, cfc). If you want to use files not processed by Railo for these special mapping directories, you have to add virtual mappings to these directories to your application server.

☐	Virtual	Resource	Archive	Primary	Inspect
	/railo-context	{railo-web}/context/	{railo-web}/context/railo-context.ra	physical	Never
☐				Resource	Always

4. Let's add a mapping to a folder that isn't in our web root; in my example, I am using `/temp/logs` but you can choose anywhere (for example, `C:\temporary\logs\` if you are using Microsoft Windows). Add the name of our mapping, `/logs`, in the text field under the **Virtual** column, and then add the path to the folder in the text field under the **Resource** column:



☐	Virtual	Resource	Archive	Primary	Inspect
	/railo-context	{railo-web}/context/	{railo-web}/context/railo-context.ra	physical	Never
☑	/logs	/temp/logs		Resource	Always

5. Now click on the **save** button to update this mapping
6. Now that we have created this mapping, let's see how we can use it in our code
7. Create a file in `<Railo Installation>/webroot/Chapter_8/` called `listing_8_05.cfm` and put the following code in there (it's nearly the same as the code in our previous *Time for action* heading):

   ```
   <cffile action="append" file="/logs/mylog.txt" output="This is the
   output to our log file! #Now()#">
   <cfloop file="/logs/mylog.txt" index="i">
     <cfoutput>#i#</cfoutput><br>
   </cfloop>
   ```

8. The difference in our code now is that the path to our log file starts with `/logs`.
9. Run the template by going to `http://localhost:8888/Chapter_8/listing_8_01.cfm` and it will still work.

What just happened

We have added a mapping to our web context that points `/logs` to another folder. This means that we can use this mapping throughout our code, and if we move our code to another server or need to change how we store the log files, all we need to do is change where the mapping points to in the Railo Server Administrator without affecting anything else. Neat eh?

Any new code that we add to our application that will need to write to the logs file now doesn't need to know where the actual path is, as we can now just reference it with the `/logs` path.

Accessing code from mappings

Reading and writing are not the only things we can do with mappings; we can do other things with these mappings. Remember components? A mapping can also be used to store our components in different locations. Let's say you want to code consistently and keep your components in a specific folder structure. In Java, they call these paths packages. Let's create some components so that we can pretend that they are part of a bigger application.

Time for action – creating our components

1. Under the `Chapter_8` folder, create another folder named `cfcs`, and then create a template called `Main.cfc` in there.

2. Put the following code in `Main.cfc`, which basically just reverses a string:

```
component output="false"{

  public function reverseIt(String input){
    return Reverse(input);
  }

}
```

3. Now that we have done that, let's call it from another template. Create a file named `listing_8_07.cfm` in the `Chapter_8` folder and put the following code inside:

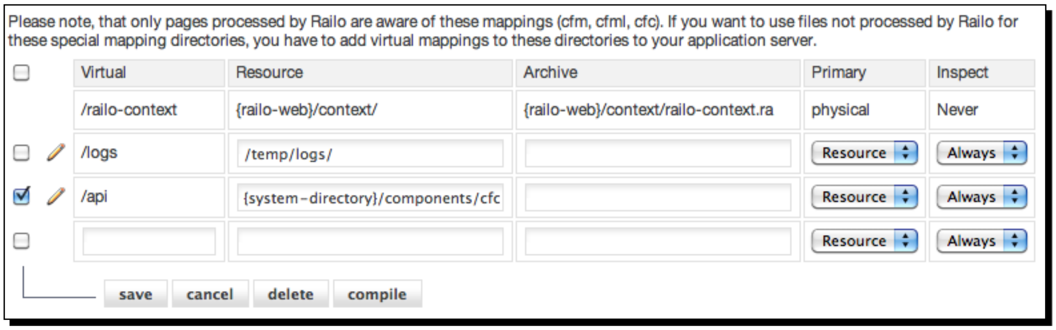
```
<cfset Main = new cfc.Main()>
<cfoutput>#Main.reverseIt("Check out my reverse!")#</cfoutput>
```

4. When we run the code `http://localhost:8888/Chapter_8/listing_8_07.cfm`, we get the following code:

```
!esrever ym tuo kcehC
```

5. Nothing new so far. But what if we don't want to keep our components accessible to the Web? It would help if we could tuck them away somewhere and only expose what we want to the Web. Let's create a folder under `<Railo Install Directory>` called `components`, and move the `cfcs` folder in there.

6. Now let's create a mapping. We are going to call it `/api` and point it to the `<Railo Install Directory>/components/cfc` directory. We can actually use one of the built-in Railo Server variables to point to our `<Railo Install Directory>`, so that we can put `{system-directory}/components/cfcs`.



Please note, that only pages processed by Railo are aware of these mappings (cfm, cfml, cfc). If you want to use files not processed by Railo for these special mapping directories, you have to add virtual mappings to these directories to your application server.

□		Virtual	Resource	Archive	Primary	Inspect
		/railo-context	{railo-web}/context/	{railo-web}/context/railo-context.ra	physical	Never
□	✎	/logs	/temp/logs/		Resource	Always
☑	✎	/api	{system-directory}/components/cfc		Resource	Always
□					Resource	Always

7. Now, click on **save** on the mapping and let's re-visit our code in the `listing_08_07.cfm` template and change the path to the main component as follows:

   ```
   <cfset Main = new api.Main()>
   <cfoutput>#Main.reverseIt("Check out my reverse!")#</cfoutput>
   ```

8. Now when we run the template, it will still work. It's reading our components from the new location.

What just happened?

From creating a component, we have been able to still access it through our mapping, so our code can remain consistent in a variety of environments, and all we need to do is change the mapping.

Once you start playing with other frameworks and libraries for CFML, you will get used to doing this. It's a good way to keep things out of the way.

Server variables

There are a number of shortcut variables that Railo Server uses, that can be used in configuration. They are:

`{railo-web}`: The path to the Railo web directory typically `{web-root}/WEB-INF/railo`

`{railo-server}`: The path to the Railo server directory typically where the `railo.jar` is located

`{temp-directory}`: The path to the temp directory of the current user of the system

`{home-directory}`: The path to the home directory of the current user of the system

`{web-root-directory}`: The path to the web root

`{system-directory}`: The path to the system directory

`{web-context-hash}`: The hash of the web context

Railo archives

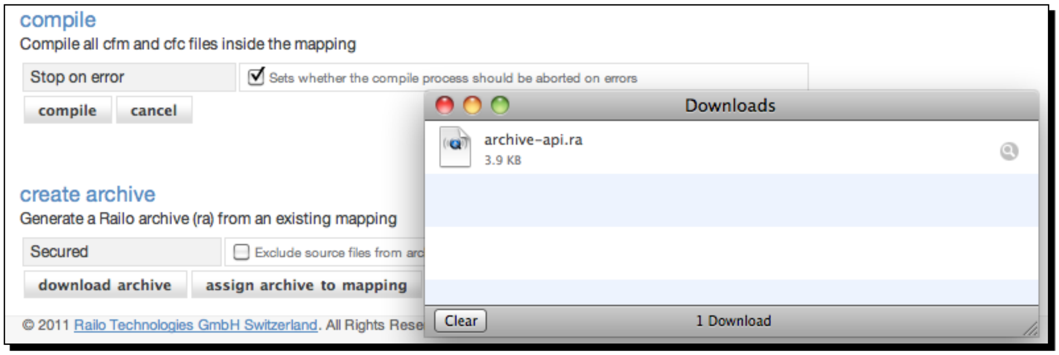
What happens if we had a number of components that we wanted to distribute to people? You could, of course, zip it up, provide instructions for people to unzip it, and so on. Also, if you want to provide code for other people to use, but you want to protect your intellectual property, you might not want to allow people to see the code itself (of course, we at Railo Technologies disagree because we are an open source project, but we understand there are cases where you wouldn't want to allow the users of your code to access it). Railo Server provides a solution with Railo Archives (`.ra`) and Secure Railo Archives (`.ras`).

Let's turn our little API into a Railo Archive.

Time for action – creating a Railo archive

Now that we have created a mapping to our API, we can easily convert it to a Railo Archive.

1. Head over to the mappings screen by going to `http://localhost:8888/railo-context/admin/web.cfm` in the Railo Web Administrator and click on the **Mappings** link under **Archives & Resources**.
2. Click on the pencil (**Edit**) icon next to the `/api` mapping to get to the **Mapping Settings** screen.
3. Scroll to the bottom of the screen to the **create archive** section.
4. You can now click on the **download archive** button, and it will give you a file called `archive-api.ra`. The file that has been generated by Railo Server is a ZIP file containing all your components, ready for you to distribute.



compile

Compile all cfm and cfc files inside the mapping

Stop on error ☑ Sets whether the compile process should be aborted on errors

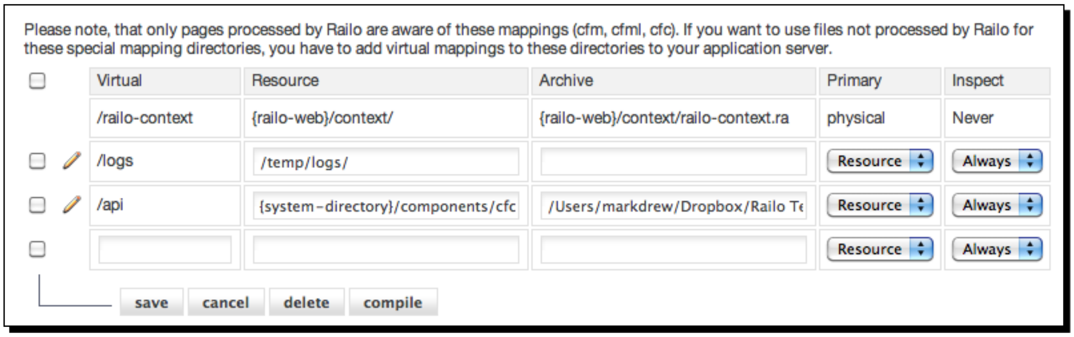
create archive

Generate a Railo archive (ra) from an existing mapping

Secured □ Exclude source files from arc

© 2011 Railo Technologies GmbH Switzerland. All Rights Rese

5. If you want to give it a test, instead of clicking on the **download archive** button, click on the **assign archive to mapping** button. This will create an archive and assign it to this mapping:



Please note, that only pages processed by Railo are aware of these mappings (cfm, cfml, cfc). If you want to use files not processed by Railo for these special mapping directories, you have to add virtual mappings to these directories to your application server.

□		Virtual	Resource	Archive	Primary	Inspect
		/railo-context	{railo-web}/context/	{railo-web}/context/railo-context.ra	physical	Never
□	✎	/logs	/temp/logs/		Resource	Always
□	✎	/api	{system-directory}/components/cfc	/Users/markdrew/Dropbox/Railo Te	Resource	Always
□					Resource	Always

What just happened?

Now that we have assigned a mapping to a folder where we keep our code, we can actually create an archive that will hold all our code. This archive can now be assigned to a mapping and the templates will be obtained from the archive. This is very useful if we want to distribute self-contained bundles of code.

Mappings and their settings

Now that we have assigned an archive to a mapping, we can remove the `components/api` folder. But before that, let's experiment with some of the settings we have in the mappings.

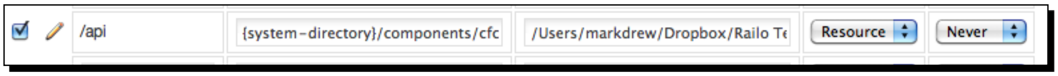
If you look in the listing of Mappings, you will see the **Primary** column. You have a choice of **Resource** or **Archive**. Currently, our `/api` mapping points both to the `components/cfcs` folder and the archive we created. Since the **Primary** setting is set to **Resource**, it means that any changes we do to our template will be reflected since that is the primary resource with a backup of the Archive.

We can actually now remove the **Resource** path and save the mapping and our code will now access the archive solely.

Let's see how this works.

Time for action – changing the settings of a mapping

In order to see what happens with the different settings assigned to a mapping, let's check that our current mapping is as follows:



☑ /api | {system-directory}/components/cfc | /Users/markdrew/Dropbox/Railo Te | Resource | Never

We have `/api` as our virtual name. We are pointing to our `{system-directory}/components/cfcs` folder and we are using an Archive that was set up for us by Railo Server. Our Primary mapping is set to **Resource**, while **Inspect** is set to **Always**.

1. Let's change the code in our `Main.cfc` component and add a new function; let's edit the component that's in our `<Railo Install Directory>/components/api/Main.cfc` and add the following:

```
component output="false"{

  public function reverseIt(String input){
    return Reverse(input);
  }
```

```
    public function getFirstLetter(String input){
      return left(input, 1);
    }

  }
```

2. Now let's add another line to the template named `listing_8_07.cfm` to call this function and add the following code:

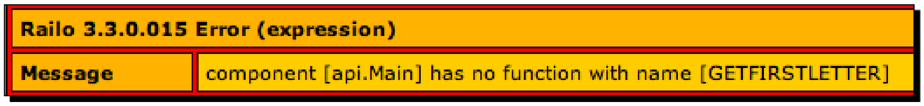
```
<cfset Main = new api.Main()>
<cfoutput>#Main.reverseIt("Check out my reverse!")#</cfoutput>
<br>
First Letter: <cfoutput>#Main.getFirstLetter("Check out my
reverse!")#</cfoutput>
```

3. When we now call the template via the browser by going to `http://localhost:8888/chapter_8/listing_8_07.cfm`, we get the following:

```
!esrever ym tuo kcehC
First Letter: C
```

4. Now let's change it so that the **Primary** mapping is using the Railo Archive we created earlier. Go back to your mappings screen, and in the `/api` mapping line, change the **Primary** select box to **Archive**.

5. When you run the code again, you get an error:



Railo 3.3.0.015 Error (expression)

Message	component [api.Main] has no function with name [GETFIRSTLETTER]

What just happened?

Since we added the code to the component that is in the mapping, and that is set as primary, Railo Server will pick up this code first. If we change the mapping to check the archive first, it then uses the archive instead. This is handy if you want to add more functionality.

As an aside, if you change **Inspect** setting to **Never**, your templates will be loaded up and cached only once during the server lifecycle. This makes it much faster to run code, as it's not checking and recompiling Railo Templates.

Accessing your files from ZIP and TAR files

A Railo Archive is a ZIP file in itself. If you were to rename it to a ZIP file, you could unzip it and see your `Main.cfc` in there. This gives you a hint that Railo Server is quite happy using zipped archives too.

Let's try this out.

Time for action – accessing files from a ZIP file

Let's create a standard header and footer for our pages, and let's say we are going to use this from all the files and we need to read it.

1. Under the `<Railo Install Directory>/webroot/Chapter_8` folder, let's create a folder named `includes`.
2. In the `includes` folder, create a couple more files, one called `header.cfm` and another called `footer.cfm`.
3. In the `header.cfm` file, let's put the following code:

```
<!DOCTYPE html>
<head>
  <title>My Page</title>
  <body id="page">
```

4. In the `footer.cfm`, let's put the following code:

```
  </body>
</head>
```

5. Now let's create a page that includes them; let's call it `listing_8_13.cfm`, save it in `<Railo Install Directory>/webroot/Chapter_8` and put the following code in the template:

```
<cfinclude template="includes/header.cfm">
  <h1>Page With Included header and footer</h1>
<cfinclude template="includes/footer.cfm">
```

6. Now when we run the template by going to `http://localhost:8888/chapter_8/listing_8_13.cfm`, we will see that we are including the header and footer. Nice!
7. Let's now zip up the `includes` folder. Here, we can use whichever ZIP program is installed on our computer. Once we have zipped it, let's name it `includes.zip` and make sure it's in the `Chapter_8` folder.

8. Now we have the two files we wanted in a ZIP file named `includes.zip`. How are we going to include them in our template now? Simple! Let's create a mapping to the ZIP file from our administrator.

9. Head back to the mappings screen in the Railo Web Administrator and add a new mapping by setting the **Virtual** name to `/includes`, and the **Resource** to `zip://<Path to Railo Install Dir>/webroot/Chapter_8/includes.zip` (for example my path is: `zip:///Users/markdrew/railoserver/webroot/Chapter_8/includes.zip`) and let's save it.

10. Let's change the code in the `listing_8_13.cfm` template to use these items in the ZIP file. Because all mappings start with `/`, the code would now look like:

    ```
    <cfinclude template="/includes/includes/header.cfm">
      <h1>Page With Included header and footer</h1>
    <cfinclude template="/includes/includes/footer.cfm">
    ```

11. When we reload the page, you will see that the page renders correctly. How neat is that?

What just happened?

Railo Server can use a number of resources including ZIP files. Here, we have created a mapping to a ZIP file that contained all the include files we wanted to use and now we can add them easily to our page.

Using RAM as a quick location to store files

As I mentioned before, filesystems can be nearly anything in Railo Server. One of the interesting uses is of RAM (Random Access Memory) as a temporary storage for templates. Why would you do this?

A good example is going back to our blog application. Imagine you have a blog post whose content you are displaying on a page, but the contents of the blog post has some code you want to execute every time someone reads the post. You would have to somehow save the content of the blog post to a file, and then include it to get it running live. But, of course, what do you do with these files all the time? Do you delete them at a certain point when the post is done? This would also mean a security issue as people could possibly access them directly (which, in all probability, you don't want them to do).

By storing them in RAM, we create a temporary resource that is available as long as we need it, and then, as part of the normal garbage collection, they will be removed.

This is where the RAM resource comes into its own.

Time for action – compiling plain text to CFML

Let's say, for example, you have a text file. One that contains some CFML code in it, but you know Railo Server won't parse it. This text could come from a database as a variable or from a file. In this example, we are just going to use a file to make it easy. Because Railo Server won't parse TXT files, this makes for a good example:

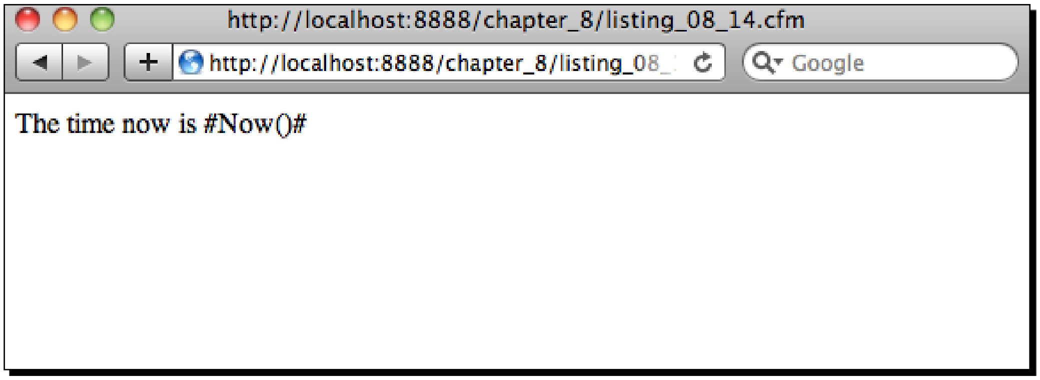
1. Create a file named `blogpost.txt` in your `Chapter_8` folder and put the following code in it:

```
The time now is <cfoutput>#Now()#</cfoutput>
```

2. Now, let's read the file and display it; let's create a template named `listing_08_14.cfm` in the `Chapter_8` folder and put the following code to display the contents of the file:

```
<cfset myBlogPost = FileRead("blogpost.txt")>
<cfoutput>#myBlogPost#</cfoutput>
```

3. When you run this template by going to `http://localhost:8888/chapter_8/listing_08_14.cfm`, you get the following displayed:



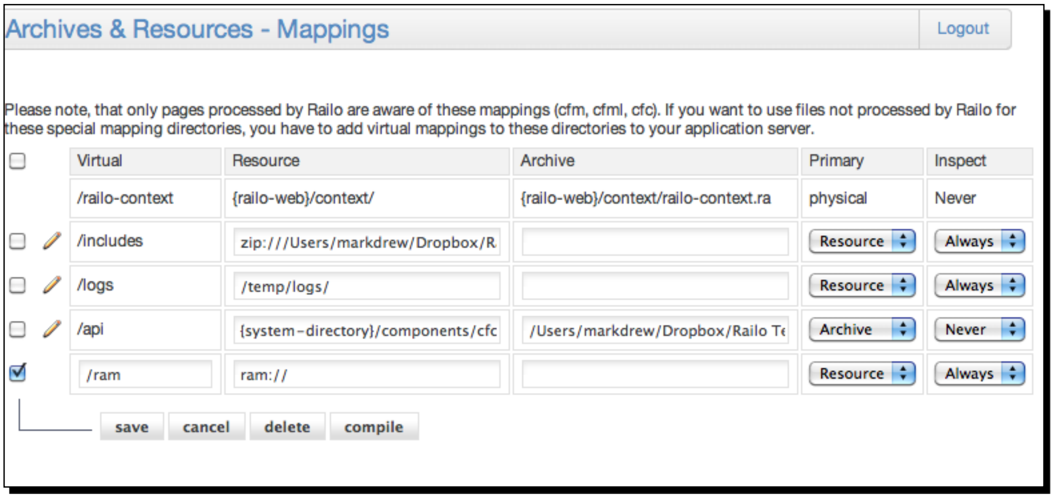
http://localhost:8888/chapter_8/listing_08_14.cfm
http://localhost:8888/chapter_8/listing_08_
Google

The time now is #Now()#

4. This is not quite what we want to be displayed. We would have expected the time to be displayed rather than the raw CFML code.

5. Let's add a RAM mapping first and see if we can get it to render on the fly.

6. Go to the **mappings** screen in the Railo Web Administrator and add a new mapping with the **Virtual** name of `/ram` and the **Resource** pointing to `ram://` and click on **Save**.



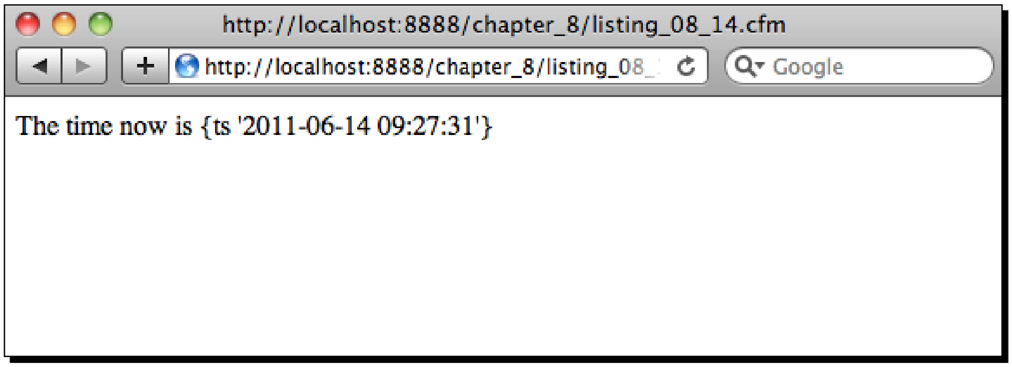
Please note, that only pages processed by Railo are aware of these mappings (cfm, cfml, cfc). If you want to use files not processed by Railo for these special mapping directories, you have to add virtual mappings to these directories to your application server.

☐		Virtual	Resource	Archive	Primary	Inspect
		/railo-context	{railo-web}/context/	{railo-web}/context/railo-context.ra	physical	Never
☐	✎	/includes	zip:///Users/markdrew/Dropbox/R		Resource	Always
☐	✎	/logs	/temp/logs/		Resource	Always
☐	✎	/api	{system-directory}/components/cfc	/Users/markdrew/Dropbox/Railo Te	Archive	Never
☑		/ram	ram://		Resource	Always

7. Now, let's copy the variable into the mapping by changing the code in the `listing_08_14.cfm` template to the following:

```
<cfset myBlogPost = FileRead("blogpost.txt")>
<cfset FileWrite("/ram/blogpost.cfm", myBlogPost)>
<cfoutput><cfinclude template="/ram/blogpost.cfm"></cfoutput>
```

8. When we run the template again in the browser, we now get the correct time and date being displayed.


http://localhost:8888/chapter_8/listing_08_14.cfm
http://localhost:8888/chapter_8/listing_08_
Google

The time now is {ts '2011-06-14 09:27:31'}

What just happened?

There are times when we need to temporarily store a file. We don't want to manage the lifecycle of this file, such as making sure we delete it after a period, but we just want to be able to instantly use it and then forget about it. This is where the RAM resource comes in handy. In the previous code, we read a variable from a text file using the `FileRead()` function (which could have been from a database or some other source) and then saved it to a virtual file in the RAM called `blogpost.cfm` using the `FileWrite()` function. Then, to display the variable, we just included this temporary file, and because it is a `.cfm` template, Railo Server will run the code for us. Impressive work so far.

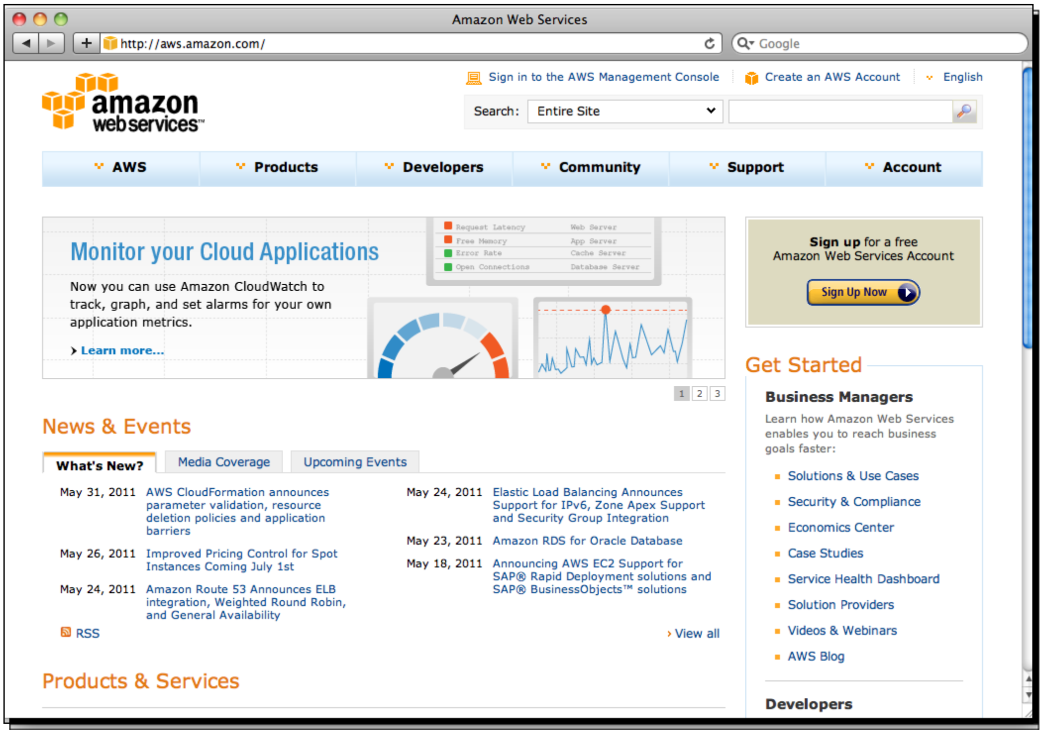
Now that we have used files from within Railo Server, how about using files that are outside of Railo Server?

Using Amazon's Simple Storage Service to use files in the Cloud

In the previous section, we looked at storing files in the temporary space of the RAM, but how about if we wanted a more permanent place to store them so that other servers might have access to them?

Amazon—the company behind the famous online store—also provides a number of web services that are very useful. One of them is their Simple Storage Service (S3). Think of it as a great way to share files, whether they are images, downloads, or videos. Railo Server can make use of S3 as yet another filesystem resource.

> For more details on Amazon Web Services, check out `https://aws.amazon.com/`



Monitor your Cloud Applications

Now you can use Amazon CloudWatch to track, graph, and set alarms for your own application metrics.

› Learn more...

Sign up for a free Amazon Web Services Account

News & Events

May 31, 2011 AWS CloudFormation announces parameter validation, resource deletion policies and application barriers

May 26, 2011 Improved Pricing Control for Spot Instances Coming July 1st

May 24, 2011 Amazon Route 53 Announces ELB integration, Weighted Round Robin, and General Availability

May 24, 2011 Elastic Load Balancing Announces Support for IPv6, Zone Apex Support and Security Group Integration

May 23, 2011 Amazon RDS for Oracle Database

May 18, 2011 Announcing AWS EC2 Support for SAP® Rapid Deployment solutions and SAP® BusinessObjects™ solutions

Products & Services

Get Started

Business Managers

Learn how Amazon Web Services enables you to reach business goals faster:

- Solutions & Use Cases
- Security & Compliance
- Economics Center
- Case Studies
- Service Health Dashboard
- Solution Providers
- Videos & Webinars
- AWS Blog

Developers

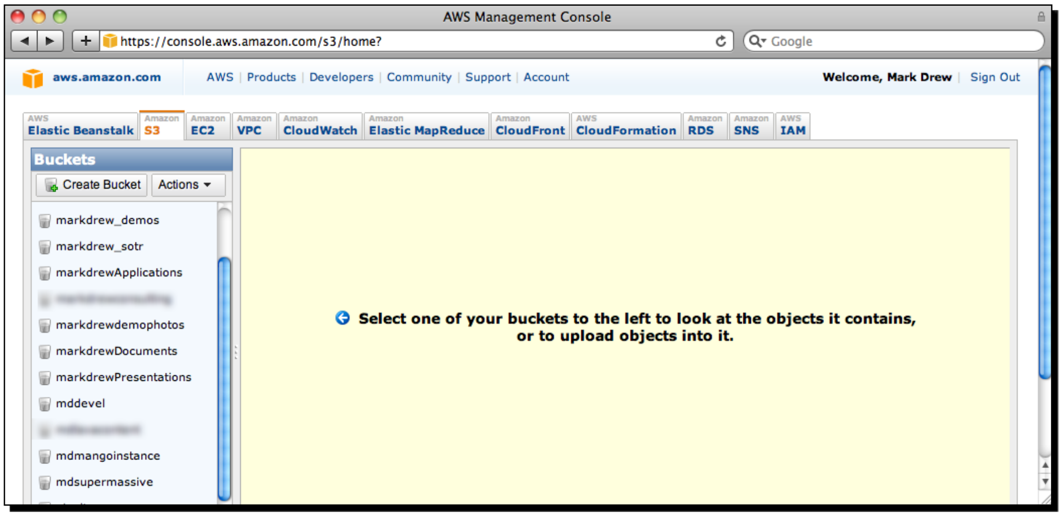
What would happen if our site got so popular that we needed to scale on demand? This is where S3 comes into its own.

If you would like to carry out the following examples, you will need to sign up for an account first, otherwise you can just walk through our examples to get an idea of how it works.

Time for action – using Amazon's Simple Storage Service (S3)

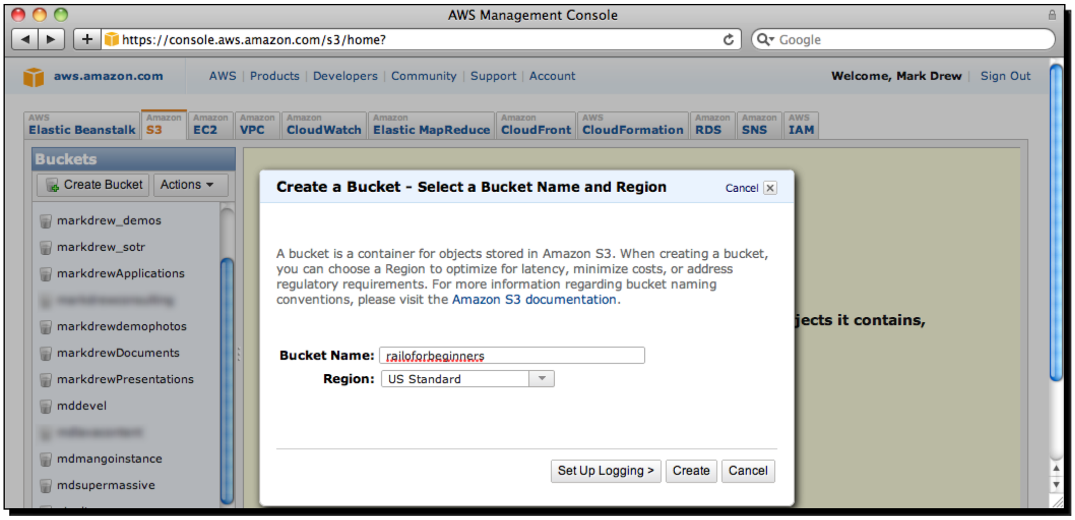
In our previous example, we used the RAM to store code we wanted to run. This works well for one server. But let's imagine we have a cluster of servers, all doing the same thing, and instead of having each server hold a copy of the same thing, we would like all of them to access these blog posts that we are rendering. All the servers would need is to have access to the same filesystem. This is where we can use an S3 bucket. An S3 bucket is a "folder" that you can assign specific settings to in the AWS (Amazon Web Services) system. Let's go and create a bucket where we are going to now store our blog posts.

Browse to `https://console.aws.amazon.com/s3/home` and log in. This is where all the "buckets" are shown in S3.



Select one of your buckets to the left to look at the objects it contains, or to upload objects into it.

1. Click on the **Create Bucket** button and enter a new bucket name. I have used **railoforbeginners** and the region as **US Standard**.



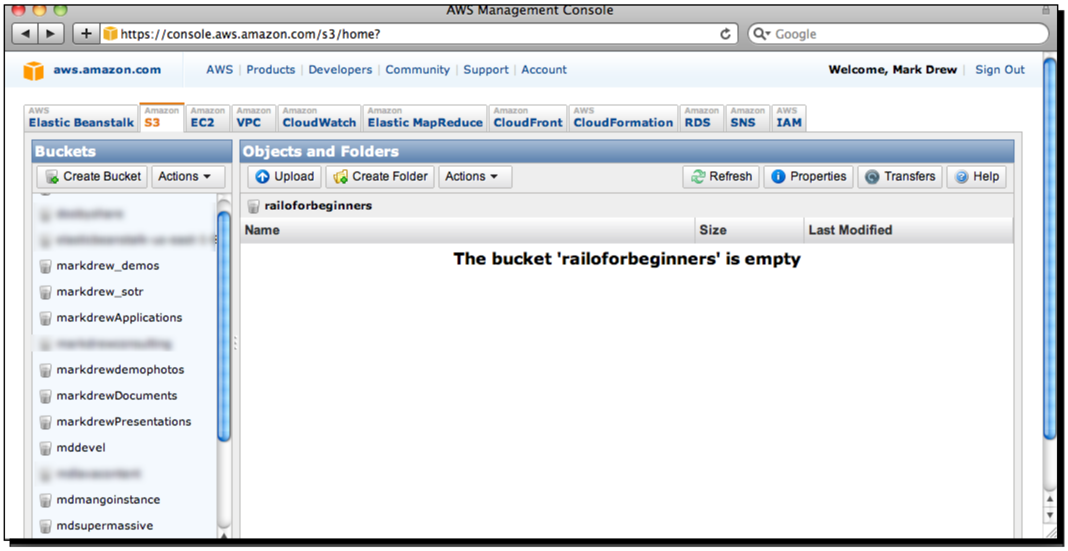
Create a Bucket - Select a Bucket Name and Region

A bucket is a container for objects stored in Amazon S3. When creating a bucket, you can choose a Region to optimize for latency, minimize costs, or address regulatory requirements. For more information regarding bucket naming conventions, please visit the Amazon S3 documentation.

Bucket Name: railoforbeginners

Region: US Standard

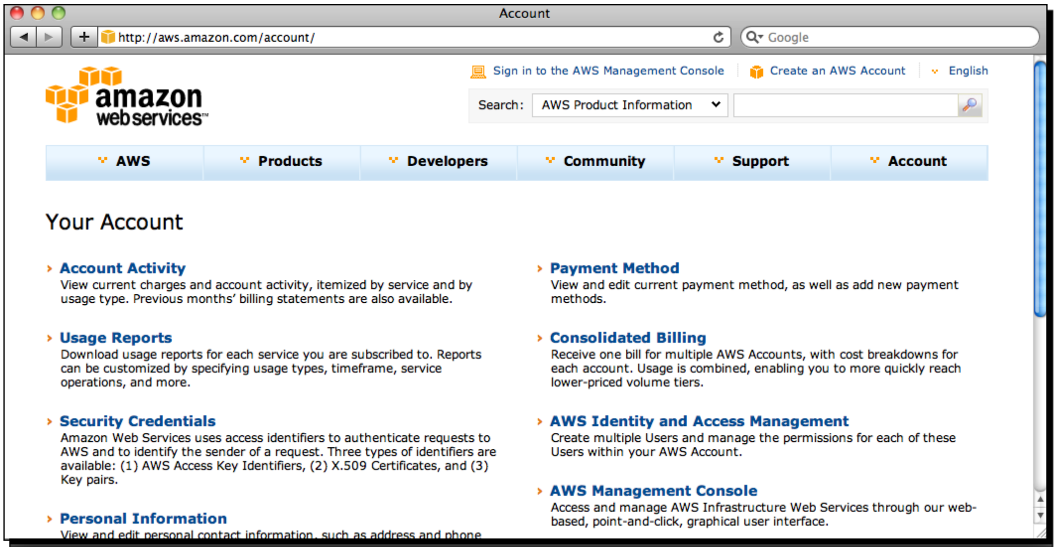
2. Click on **Create** and your bucket will be created:



The bucket 'railoforbeginners' is empty

3. Now, before we can create a mapping to this S3 bucket, we need to get our Amazon Access Credentials. That is, our **Access Key ID** and our **Secret Access Key**. They are used by Railo Server to connect to the bucket, proving that you have the rights to read and write to the bucket. Think of them as your username and password.

4. From the AWS console, click on the **Account** link at the top:



Your Account

Account Activity
View current charges and account activity, itemized by service and by usage type. Previous months' billing statements are also available.

Usage Reports
Download usage reports for each service you are subscribed to. Reports can be customized by specifying usage types, timeframe, service operations, and more.

Security Credentials
Amazon Web Services uses access identifiers to authenticate requests to AWS and to identify the sender of a request. Three types of identifiers are available: (1) AWS Access Key Identifiers, (2) X.509 Certificates, and (3) Key pairs.

Personal Information
View and edit personal contact information, such as address and phone

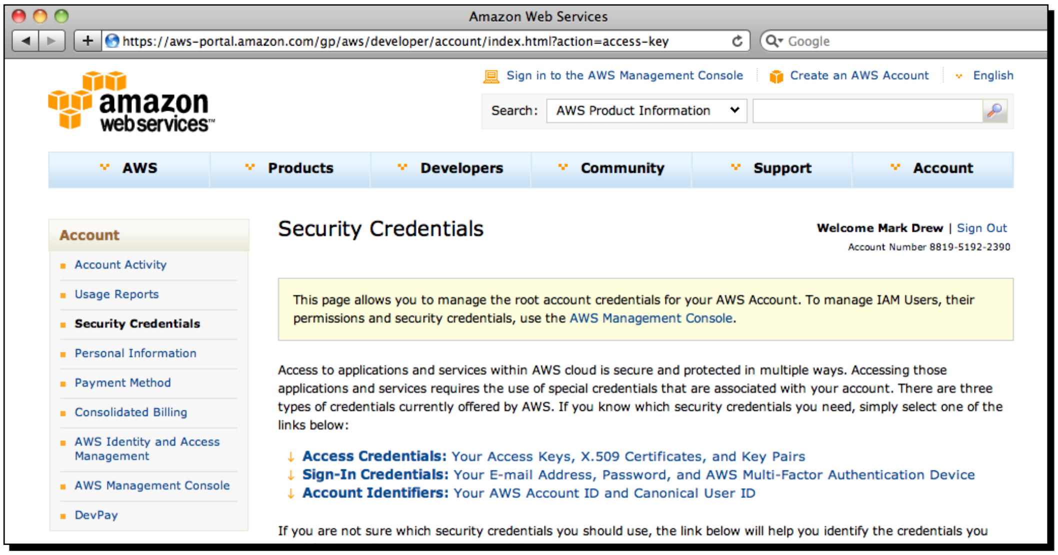
Payment Method
View and edit current payment method, as well as add new payment methods.

Consolidated Billing
Receive one bill for multiple AWS Accounts, with cost breakdowns for each account. Usage is combined, enabling you to more quickly reach lower-priced volume tiers.

AWS Identity and Access Management
Create multiple Users and manage the permissions for each of these Users within your AWS Account.

AWS Management Console
Access and manage AWS Infrastructure Web Services through our web-based, point-and-click, graphical user interface.

5. Then click on the **Security Credentials** link.



Security Credentials

Welcome Mark Drew | Sign Out
Account Number 8819-5192-2390

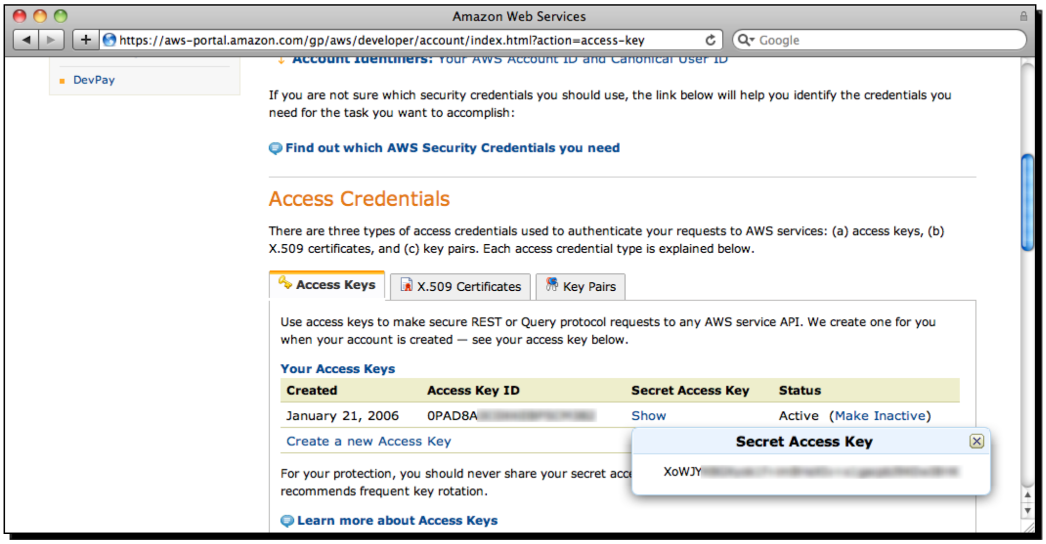
This page allows you to manage the root account credentials for your AWS Account. To manage IAM Users, their permissions and security credentials, use the AWS Management Console.

Access to applications and services within AWS cloud is secure and protected in multiple ways. Accessing those applications and services requires the use of special credentials that are associated with your account. There are three types of credentials currently offered by AWS. If you know which security credentials you need, simply select one of the links below:

↓ **Access Credentials:** Your Access Keys, X.509 Certificates, and Key Pairs
↓ **Sign-In Credentials:** Your E-mail Address, Password, and AWS Multi-Factor Authentication Device
↓ **Account Identifiers:** Your AWS Account ID and Canonical User ID

If you are not sure which security credentials you should use, the link below will help you identify the credentials you

6. In the next page, scroll down to the **Access Credentials** section, copy the **Access Key ID**, and click on the **Show** link under the **Secret Access Key** to show it and copy these somewhere.



Amazon Web Services
https://aws-portal.amazon.com/gp/aws/developer/account/index.html?action=access-key
Google
↓ Account Identifiers: Your AWS Account ID and Canonical User ID
DevPay

If you are not sure which security credentials you should use, the link below will help you identify the credentials you need for the task you want to accomplish:

Find out which AWS Security Credentials you need

Access Credentials

There are three types of access credentials used to authenticate your requests to AWS services: (a) access keys, (b) X.509 certificates, and (c) key pairs. Each access credential type is explained below.

Use access keys to make secure REST or Query protocol requests to any AWS service API. We create one for you when your account is created — see your access key below.

Your Access Keys

Created	Access Key ID	Secret Access Key	Status
January 21, 2006	0PAD8A[redacted]	Show	Active (Make Inactive)

Create a new Access Key

Secret Access Key

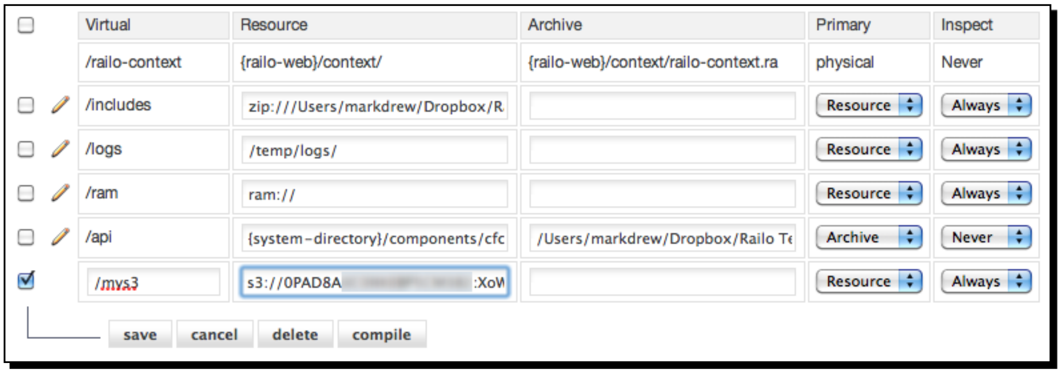
XoWJY[redacted]

For your protection, you should never share your secret acc recommends frequent key rotation.

Learn more about Access Keys

7. Now, we are going to go back to the **Mappings** page in the Railo Web Administrator and add a mapping that points to our bucket.

8. Enter `/mys3` in the **Virtual** mapping name box. Then we are going to enter the **Resource** that points to our S3 bucket. It is in the format `s3://<Access Key ID>:<Secret Access Key>@<bucketname>/`, so for my example, I have `s3://0PAD8A------:XoWJY--------@railoforbeginners/` (I have removed the full string as obviously this is my account.):



☐		Virtual	Resource	Archive	Primary	Inspect
		/railo-context	{railo-web}/context/	{railo-web}/context/railo-context.ra	physical	Never
☐	✎	/includes	zip:///Users/markdrew/Dropbox/R		Resource	Always
☐	✎	/logs	/temp/logs/		Resource	Always
☐	✎	/ram	ram://		Resource	Always
☐	✎	/api	{system-directory}/components/cfc	/Users/markdrew/Dropbox/Railo Te	Archive	Never
☑		/mys3	s3://0PAD8A[redacted]:XoW		Resource	Always

9. Now that you have entered your access key and path to the bucket, you can click **save**.

10. Let's now create the same functionality we had with our RAM example but using S3.

11. Create a template in the `Chapter_8` folder named `listing_08_16.cfm` and put the following code (it's very similar to the RAM example):

```
<cfset myBlogPost = FileRead("blogpost.txt")>
<cfif NOT FileExists("/mys3/blogpost.cfm")>
<cfset FileWrite("/mys3/blogpost.cfm", myBlogPost)>
  Writing file to S3<br>
</cfif>

<cfoutput><cfinclude template="/mys3/blogpost.cfm"></cfoutput>
```

12. When we run our template by going to `http://localhost:8888/chapter_8/listing_08_16.cfm`, we should get the same code as before.

What just happened?

Mappings in Railo Server can use a number of resources that can be treated like filesystems, even remote ones such as Amazon's S3 service. In the previous code, we created a mapping as usual but pointed it to our S3 bucket.

Going through the code, you see that we read the contents of `blogpost.txt` as a variable. We then check to see if we have a `blogpost.cfm` on our `/mys3` mapping, which is pointing to our S3 bucket. If the file doesn't exist, we can create it and show that we are writing to it. Then we include the file, again from the S3 bucket.

This means all our servers will have access to the file, and if they aren't there, one of the servers will create it for us. Pretty neat!

Summary

Hopefully, this chapter has given you a good idea about resources and mappings in Railo Server.

We covered:

- Reading and writing to local files with the `<cfile>` tag
- Easily looping over the contents in a file using the `<cfloop file="">` tag
- Creating mappings to directories in our filesystem and how to access the templates and components
- Creating Railo Archives from mappings, using them, and overriding the order, depending on whether a resource or an archive is used
- Using ZIP files as other Archives in our mappings
- Using RAM as a mapping to compile and render Railo Templates
- Using Amazon's Simple Storage Service as a filesystem to access our files from other Railo Server instances.

Now that you understand mappings, we can move onto the next chapter, in which we extend the functionality of a Railo Server.

9

Extending Railo Server

Now that you know all the ins and outs of using CFML and the existing functionalities of Railo Server, it is time to go one magnificent step further. You are not limited by any boundaries in Railo Server; you can easily push those boundaries yourself. Welcome to the world of extending Railo Server.

In this chapter, we shall:

- Create a new CFML tag and function
- Install a Railo Extension
- Create a Railo Application Extension
- Create a Railo Server Extension
- Develop and deploy our own Extension Provider

There's much to do, so let's get started!

Why create your own CFML tags and functions?

You might ask yourself why you should do this. Isn't it bad practice to be making your own custom additions to an existing scripting language? Well, there are a lot of reasons why Railo Server gives you this option.

First off, Railo Server does not include the exact same CFML functionality as Open BlueDragon or Adobe ColdFusion®. About 95 percent is the same, but you can find some differences and un-implemented features. By having the option of creating (and overwriting) CFML tags and functions, you can change the workings of Railo Server by yourself to make it suit your needs.

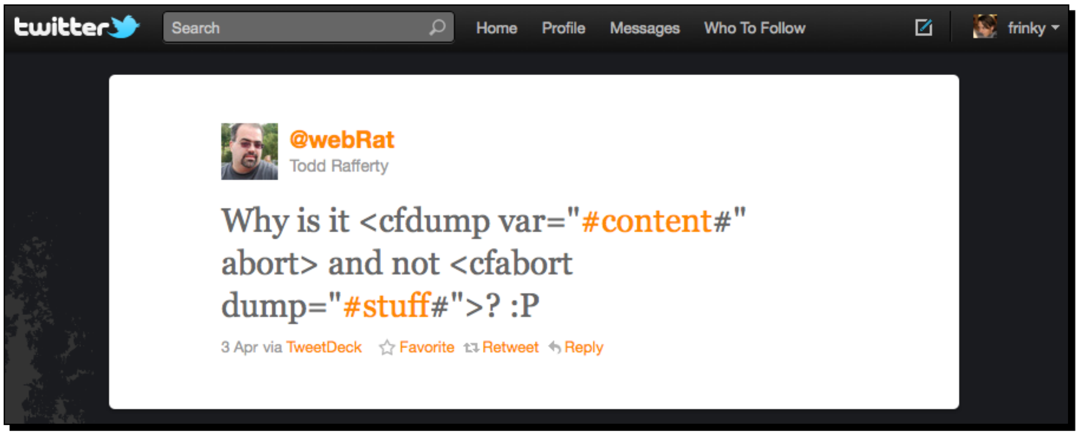
By adding your own CFML tags and functions, you can write more of your code in the same coding style. This will make it much easier to read and understand, especially when comparing it to using includes or CFC function calls.

Another great reason is the ability to use this new functionality not just for you, but for the whole Railo community. It is as easy as adding it to the Railo Extension Store, or to distribute it via your own Extension provider, as we shall see shortly.

> Custom CFML tags and functions are written in CFML. This means that you don't have to know any Java, C++, or other complicated stuff. We can just keep it simple, as we shall see.

Time for action – creating our own CFML tag

One of the Railo team members, Todd Rafferty, tweeted the following a while ago:



@webRat
Todd Rafferty

Why is it <cfdump var="#content#" abort> and not <cfabort dump="#stuff#">? :P

3 Apr via TweetDeck ☆ Favorite Retweet Reply

I thought he had a good point there, so let's take this example and create the functionality for him. Heck, we could even propose our enhanced tag to be added into the Railo core!

1. Open up your editor and write the following code:

```
<cfcomponent name="abort" output="false">

  <cfset this.metadata.attributetype="fixed" />
  <cfset this.metadata.attributes = {
    showerror:  {required:false, type: "string"}
    , dump:    {required:false, type: "any"}
  } />
  <cffunction name="init" output="false" returntype="void"
hint="invoked after tag is constructed">
    <cfargument name="hasEndTag" type="boolean" required="true" />
    <cfargument name="parent" type="component" required="false"
hint="the parent cfc custom tag, if there is one" />
  </cffunction>

  <cffunction name="onStartTag" returntype="boolean">
    <cfargument name="attributes" type="struct" required="true" />
    <cfargument name="caller" type="struct" required="true" />

    <cfif structKeyExists(arguments.attributes, "dump")>
      <cfdump var="#arguments.attributes.dump#" label="dump via
cfabort" />
    </cfif>


    <!--- Create an instance of the original Railo Abort tag --->
    <cfset var abortJavaObject = createObject("java", "railo.
runtime.tag.Abort") />
    <!---  set the error text when given --->
    <cfif structKeyExists(arguments.attributes, "showerror")>
      <cfset abortJavaObject.setShowerror( javaCast("string",
arguments.attributes.showerror) ) />
    </cfif>
    <!---  call the cfabort function --->
    <cfset abortJavaObject.doStartTag() />

    <cfreturn true />
  </cffunction>

  <cffunction name="onEndTag" output="true" returntype="boolean">
    <cfargument name="attributes" type="struct" required="true" />
    <cfargument name="caller" type="struct" required="true" />
      <cfargument name="generatedContent" type="string"
required="false" />
    <cfreturn false />
  </cffunction>
</cfcomponent>
```

2. Save this file to the tag library, for example:

    ```
        Tomcat:<Railo install directory>/tomcat/railo/railo-
    server/context/library/tag/
    Railo Express: Railo install directory>/lib/ext/railo-server/
    context/library/tag/
    ```

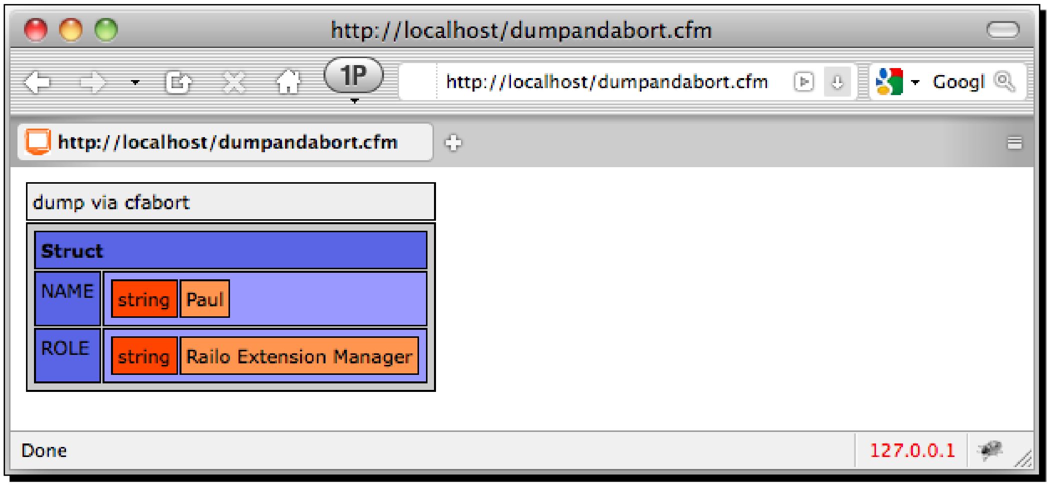
3. Save this file as `Abort.cfc` in `<Railo install directory>/tomcat/railo/railo-server/context/library/tag/`, or `<Railo install directory>/lib/ext/railo-server/context/library/tag/`, if you are using Railo Express edition.

4. Now, restart your Railo Server instance for this new tag to be picked up by clicking on the **Restart** link in the Server Administrator navigation or by running the following code:

    ```
    <cfadmin action="reload" type="server" password="Your-server-
    admin-pasword" />
    ```

5. Let's create a new CFML file `dumpandabort.cfm` with the following content:

    ```
    <cfset myData = {name: "Paul", role: "Railo Extension Manager"} />
    <cfabort dump="#myData#" />
    ```


http://localhost/dumpandabort.cfm
1P
http://localhost/dumpandabort.cfm
Googl
http://localhost/dumpandabort.cfm
Done
127.0.0.1

```
dump via cfabort
Struct
NAME  string  Paul
ROLE  string  Railo Extension Manager
```

What just happened?

In short, we changed the working of the `<cfabort>` tag in our Railo instance.

First, we created the `Abort.cfc` file. We could have made this CFC even smaller by removing the argument declarations in the `init` and `onEndTag` functions. I left them there, so that you can see which arguments are given to those functions.

The function onStartTag within the CFC contains all our functionality. If we take a closer look at that code, we can see that we first check if a dump attribute was given in the <cfabort> tag. If so, we do a <cfdump> with the given data:

```
<cfif structKeyExists(arguments.attributes, "dump")>
  <cfdump var="#arguments.attributes.dump#" label="dump via cfabort"
/>
</cfif>
```

Now all that's left to do is to make sure that a regular <cfabort> action is executed. But how do we do this? We can't simply call <cfabort>, because that would once again trigger our newly created Abort.cfc, this would cause an infinite loop, and would give you both a headache and a non-responding computer. So, let's move on to a better solution.

You know that Railo Server is an open source product, right? That means we can download and inspect the source code. For this exercise, I searched for "**abort**" within the Railo source code, which showed me the following file:

https://github.com/getrailo/railo/blob/master/railo-java/railo-core/src/railo/runtime/tag/Abort.java. Because it is stored in the directory /railo/runtime/tag/, this is most definitely the file we need.

I will not get into Java specifics here, especially because I haven't got the faintest clue as to how to write Java code myself. But from looking at the source code, I could extract the following pieces of code within that file:

```
package railo.runtime.tag;
...
public final class Abort extends TagImpl {
...
    public void setShowerror(String showerror)  {
        this.showerror=showerror;
    }
...
  public int doStartTag() throws PageException      {
    if(showerror!=null) throw new AbortException(showerror);
    throw new railo.runtime.exp.Abort(type);
  }
...
}
```

Combining the Java package name with the filename, we get `railo.runtime.tag.Abort` as the class name:

```
<cfset var abortJavaObject = createObject("java", "railo.runtime.tag.
Abort") />
```

Because there is another optional parameter for the `<cfabort>` tag, `showerror`, we need to set that attribute in the `abort` object we just created:

```
<cfif structKeyExists(arguments.attributes, "showerror")>
  <cfset abortJavaObject.setShowerror( javaCast("string", arguments.
attributes.showerror) ) />
</cfif>
```

> Because we are directly dealing with a Java object here, we need to make sure that we are sending the correct argument type to the function `setShowError`. In this case, that would be a string, and hence we need `javaCast("string", arguments.attributes.showerror)`.

Now that we got the Java object ready for action, we can simply run the original abort function like this:

```
<cfset abortJavaObject.doStartTag() />
```

After writing this code, we saved it into `<Railo>..../context/library/tag/`.

As the directory name suggests, we got a `library` available to us where we can save our custom tags and functions. You can also find this library path within the `WEB-INF` directory of each web context, where you can save custom tags and functions to be used only for the web context.

CFML functions

When I say *CFML functions*, you might think of the built-in functions such as `isDefined()` or `trim()`, but you might also think about `<cffunction name="myFunction">`. The main difference between the two is that the latter needs to be explicitly set within your application before it can be used, while the built-in functions can just be called from anywhere at any time.

What we are going to do now is convert our own function into a built-in CFML function. Let's start!

Time for action – creating our own CFML function

Now that we know how to create our own CFML tag in Railo Server, it won't be very hard to create our own CFML function. And indeed, it isn't!

Let's create the function `cleanScope()`, which cleans the contents of a scope like URL or form (or any CFML structure actually). This could save us some lines of code in our next project:

1. Create a file with the following content:

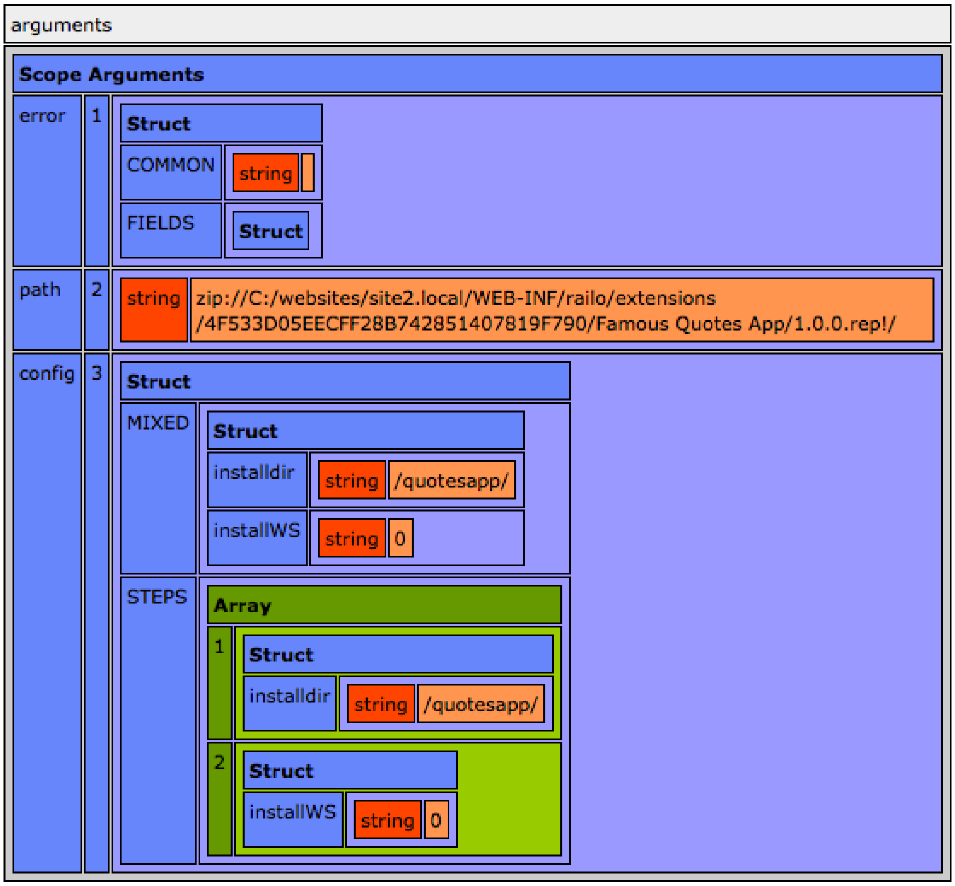
```
<cffunction name="cleanscope" output="false" access="public"
returntype="any" hint="I clean a given struct/array from spaces
and script injection">
  <cfargument name="scope" type="any" required="true" hint="The
scope to clean (e.g. URL of form)" />
  <cfset var key = "" />
  <cfif not isStruct(arguments.scope) and not isArray(arguments.
scope)>
    <cfthrow message="The argument for function cleanscope must be
either a struct or array!" />
  </cfif>

  <cfloop collection="#arguments.scope#" item="key">
    <!--- if the value is a simple value (string, number, etc.)
--->
    <cfif isSimpleValue(arguments.scope[key])>
      <!---  replace any instance of "<script", "</script", or
"<iframe", with "<cleaned" --->
      <cfset arguments.scope[key] = rereplaceNoCase(arguments.
scope[key], "<(/?)(script|iframe)", "<\1cleaned", "all") />
      <!---  remove spaces, tabs and line feeds from the start and
end of each value --->
      <cfset arguments.scope[key] = trim(arguments.scope[key]) />
    <!--- If we find an array or struct, we will clean those
contents too --->
    <cfelseif isStruct(arguments.scope[key]) or isArray(arguments.
scope[key])>
      <cfset arguments.scope[key] = cleanscope(arguments.
scope[key]) />
    </cfif>
  </cfloop>
  <cfreturn arguments.scope />
</cffunction>
```

2. Save the file as `cleanscope.cfm` in `<Railo install directory>/tomcat/railo/railo-server/context/library/function/`, or `<Railo install directory>/lib/ext/railo-server/context/library/function/`, if you are using the Railo Express edition.

3. Now create a test file named `cleanScopeTest.cfm` with the following content:

```
<!--- first off, let's restart Railo Server for the new custom
function to be picked up (we only need to do this one time) --->
<cfadmin action="reload" type="server" password="Your-server-
admin-pasword" />

<!---  if the form scope is not empty, clean it --->
<cfif structCount(form) gt 0>
  <cfset cleanscope(form) />
  <!---  dump the form values --->
  <cfdump eval=form />
</cfif>
<form method="post" action="cleanscopetest.cfm">
  <input type="text" name="evilvalue" value="&lt;script>alert('boo
')&lt;/script>" size="50" /><br />
  <input type="text" name="spacy" value=" Hi  there! " size="50"
/><br />
  <input type="text" name="name.first" value="  Railo  " size="50"
/><br />
  <input type="text" name="name.last" value="    Server
"size="50" /><br />
  <input type="text" name="versions[]" value=" 3.3 " size="50"
/><br />
  <input type="submit" value="test" />
</form>

<cfabort />
```

4. When we save this file to the `<Railo Install Directory>/webroot`, run it, and click on the **test** button, we get the following output:



```
arguments
Scope Arguments
error   1  Struct
           COMMON  string
           FIELDS  Struct
path    2  string  zip://C:/websites/site2.local/WEB-INF/railo/extensions
                   /4F533D05EECFF28B742851407819F790/Famous Quotes App/1.0.0.rep!/
config  3  Struct
           MIXED   Struct
                   installdir  string  /quotesapp/
                   installWS   string  0
           STEPS   Array
                   1  Struct
                      installdir  string  /quotesapp/
                   2  Struct
                      installWS   string  0
```

What just happened?

Just as we previously created a CFML tag, we now created and tested an internal CFML function. The only thing we needed to do was define the `<cffunction>` and save it to the `library/function/` directory of Railo Server.

There are a few caveats that you need to be aware of when creating custom CFML functions:

- The function name and the filename must be exactly the same (`cleanscope` in the example).
- The file extension must be `.cfm`, and not `.cfc` as you might expect.
- When using a function written in `<cfscript>`, it must be surrounded by a `<cfscript>` tag.

Using return type "any"

Did you notice the value *any* (on the first line) for the `returntype` and `cfargument type` within the function? The reason we used this type has to do with the fact that we needed to allow both a structure and an array as the argument. Because we return the argument at the end of the function (`<cfreturn arguments.scope />`), we also needed to set the `returntype` to `any`. This setting makes all types of incoming data possible, so we needed to add an extra check:

```
<cfif not isStruct(arguments.scope) and not isArray(arguments.scope)>

  <cfthrow message="The argument for function cleanscope must be
either a struct or array!" />
</cfif>
```

Of course, it is your own decision to include these kinds of checks, but if you hate debugging strange errors as much as I do, well, then you'd better leave it where it is.

Structure and array notation in the form and URL scope

If you take a closer look at the example page we created, you will see the input names `versions[]`, `name.first`, and `name.last`. If you are coming from the PHP world, then you are probably familiar with this syntax, but if you're not, keep on reading.

The **array notation** `versions[]` creates an array called `versions` in the `form` scope, where the value of the form variable is used for the array value. If you want to add another value to this array, then you need to send another form variable called `versions[]`.

For example, let's have a look at the following lines of code:

```
<input name="versions[]" value="3.2" type="text" />
<input name="versions[]" value="3.3" type="text" />
```

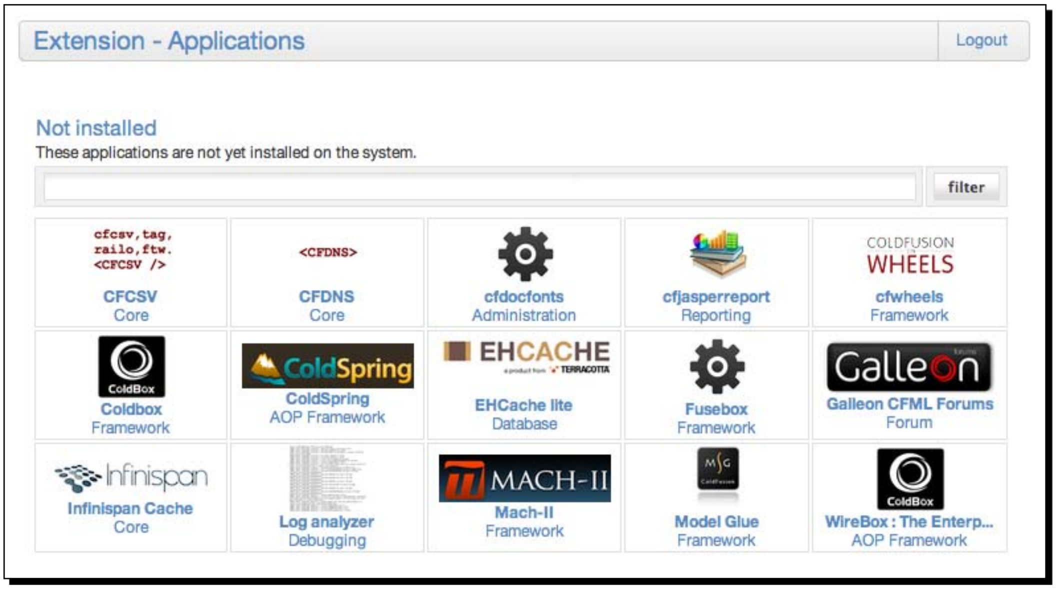
This code will create an array called `versions`, with the values `3.2` and `3.3`. Setting a list as the input value, that is, `3.2,3.3`, does *not* create two items in the array. You really need to send two form variables.

The **dot notation** `name.last` creates a structure within the form scope. As you can see in the previous image, it created a structure called `name`, with the keys `first` and `last`. Besides this structure, these form variables are *also* available by their original names, `name.last` and `name.first`.

This is a new notation, which can help you in pre-organizing your FORM or URL scope. On the other hand, you need to take this into account when you are working with the FORM or URL scope; you might encounter non-simple values where you did not expect them to be.

Installing extensions

Have you taken a look at the **Extension** | **Applications** menu item in the Web and Server administrator yet? It contains tons of useful applications, eagerly waiting to be installed by you:



Extension - Applications

Not installed

These applications are not yet installed on the system.

CFCSV Core	CFDNS Core	cfdocfonts Administration	cfjasperreport Reporting	cfwheels Framework
Coldbox Framework	ColdSpring AOP Framework	EHCache lite Database	Fusebox Framework	Galleon CFML Forums Forum
Infinispan Cache Core	Log analyzer Debugging	Mach-II Framework	Model Glue Framework	WireBox : The Enterp... AOP Framework

The previous screenshot shows the list of extensions available in the Railo Web Administrator.

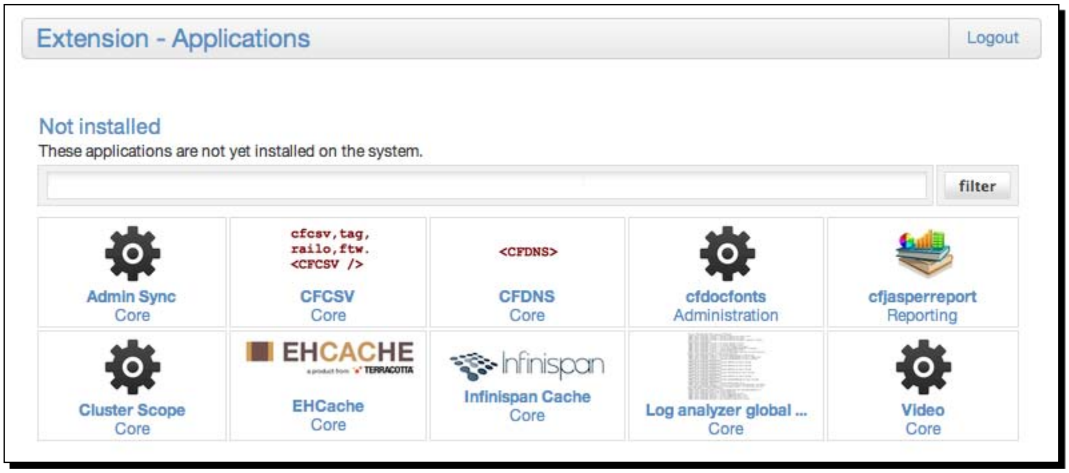
These applications are generally standalone applications that can be installed in the context that the Railo Web Administrator is being used in. For example, we have various frameworks and applications available to us. When you install one of these applications, it will only be available in the context that you have installed it into and not affect other parts of the server.

But what about applications that are not context-specific, for example, if you want to extend the capabilities of the whole server?

These applications are installed from the Railo Server Administrator. They are applications that change or add some behavior to the whole server, and that include all the other contexts that are running on that server.

Some examples of these types of applications are:

- **Admin Sync**: This application allows you to synchronize the settings of one server with another completely separate server.
- **Cluster Scope**: This provides a new scope named CLUSTER that can be read between different servers in a cluster.
- **EHCache Core**: This extension allows you to install a fully featured and clusterable version of the EHCache caching server.



Extension - Applications

Not installed

These applications are not yet installed on the system.

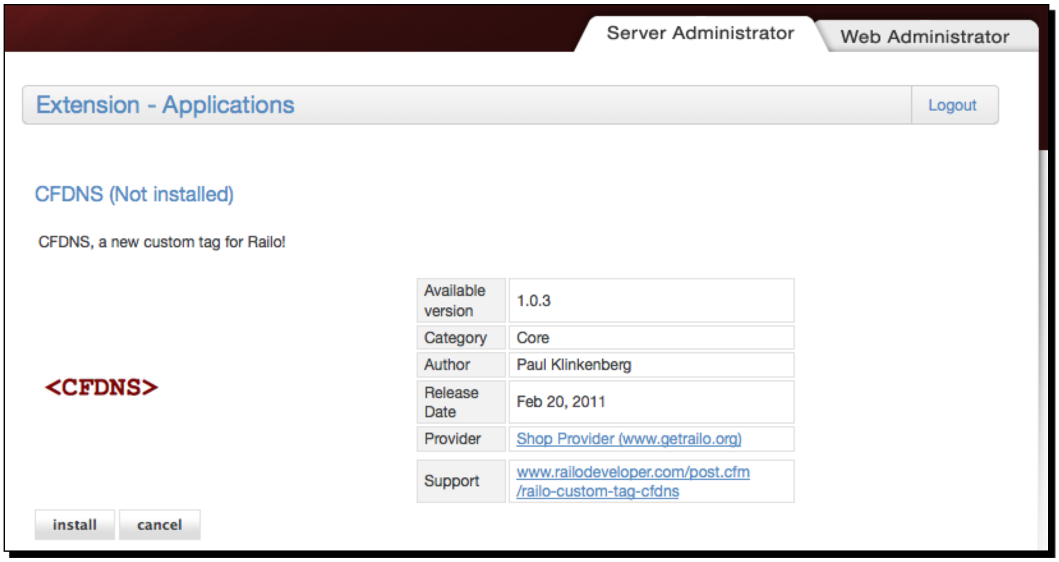
This screenshot displays the list of extensions available in the Railo Server Administrator.

Some of them are paid extensions, others are free. Let's start by installing a new custom tag: `<cfdns>`.

Time for action – installing an extension for the web context

As the title suggests, we're going to install an extension for one web context. This needs to be done via the corresponding web administrator.

1. Go to `http://localhost:8080/railo-context/admin/web.cfm` (or browse as according to your local setup, for example, `site2.local:8888`).
2. After logging in, go to the menu item **Extension** | **Applications**.
3. Under the heading **Not installed**, click on the link **CFDNS**.
4 You will get a details page with the specifications of the extension:



Extension - Applications

CFDNS (Not installed)

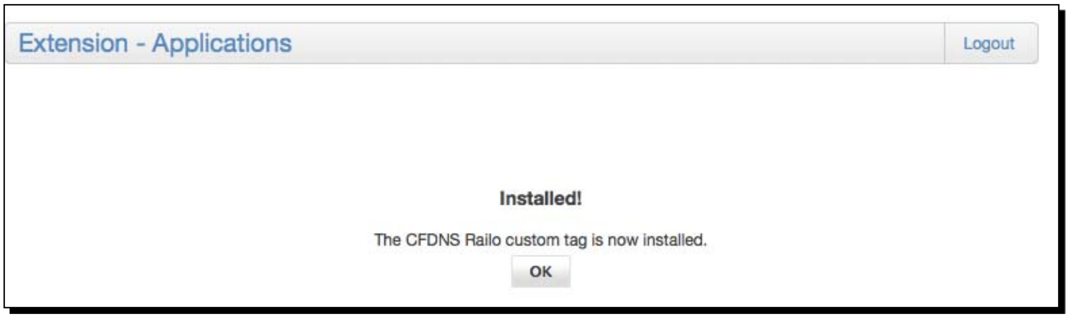
CFDNS, a new custom tag for Railo!

<CFDNS>

Available version	1.0.3
Category	Core
Author	Paul Klinkenberg
Release Date	Feb 20, 2011
Provider	Shop Provider (www.getrailo.org)
Support	www.railodeveloper.com/post.cfm /railo-custom-tag-cfdns

5. Now click on **Install**.

6. After agreeing to the license information, you'll get the following screen:



Extension - Applications

Installed!

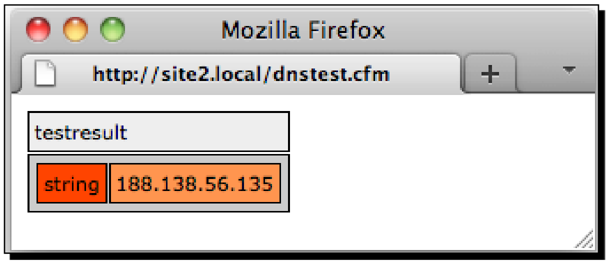
The CFDNS Railo custom tag is now installed.

7. That screen means you have installed a new Railo Server custom tag within 30 seconds.

8. Let's test this by creating a new file `dnstest.cfm` with the following content:

```
<cfdns action="getaddress" host="www.getrailo.org"
variable="testresult" />
<cfdump eval=testresult />
```

9. When we save that page and run it, we get the following output:



testresult

string 188.138.56.135

What just happened?

By using the Railo Web Administrator, we installed a Railo custom tag for use in our web context. This means we can use it for the website we are on (`localhost:8080`), but not in other web contexts like `site2.local`.

We also tested whether the new tag worked, by doing an IP lookup for `http://www.getrailo.org`.

If you want more information about the working of the `<cfdns>` tag, just visit `http://www.railodeveloper.com/post.cfm/railo-custom-tag-cfdns`.

Server versus web extensions

The look and feel of both Railo admins is exactly the same, and the procedure for installing extensions is also exactly the same. However, there is one difference between the two, namely, the list of available extensions.

There are extensions that are only useful when installed in a specific web context, for example, the installation of a blog application. A typical Server extension is the (paid) **Admin-synchronization extension**. This extension keeps the settings for multiple Railo installations in sync.

There is a third category of extensions, and it is available in both Administrators. The CFDNS extension, for example, is available in both admins. If you have your Railo website hosted in a shared-hosting environment, then you will probably have access to your own web admin, but not to the global server admin. The great thing about Railo is that you don't have to open a ticket with your hosting provider to have the extension added (or updated), as you can just manage it locally with your own web admin.

The extension installation system

We have gone through the installation of the CFDNS extension. We clicked it in the admin, hit the **Install** button, agreed to the license terms, and we were done. It doesn't get more basic than that.

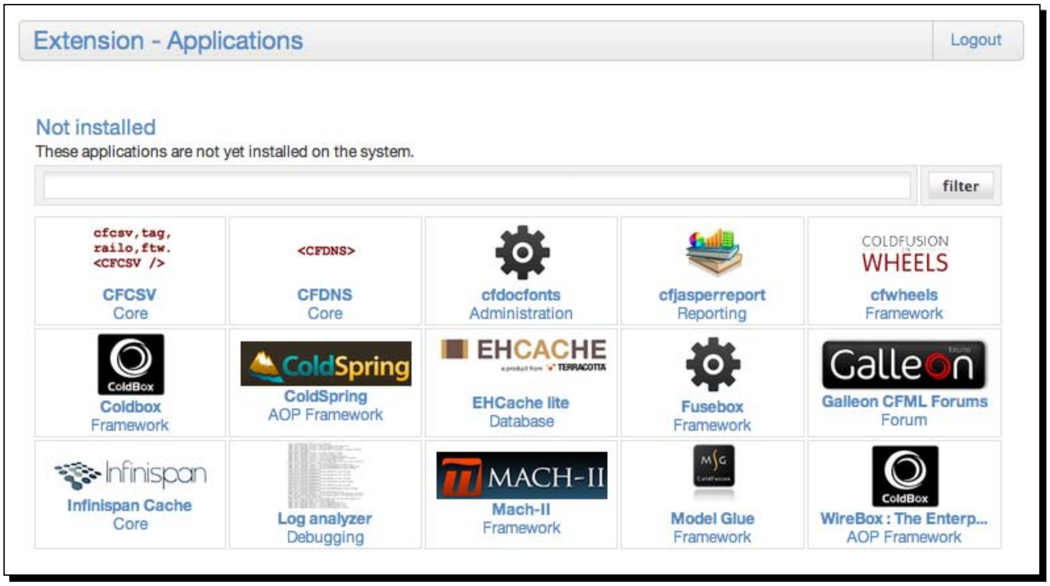
The extension installation system offers us much more, with data input and validation options across multiple steps.

Time for action – installing the Galleon forums web application

In order to see a more complex application installed, we are going to install the Galleon forums application in our web context:

1. Before we go ahead, Galleon needs a database to store all of its information. We need to create a database and assign a datasource to it (we have covered this already a number of times; just create a new database in your MySQL database server and a datasource in the **Services** | **Datasource** section of the administrator) and call it **Galleon** to keep it consistent.

2. Let's go and check out the available applications in our Railo Web Administrator by going to `http://localhost:8888/railo-context/admin/web.cfm` and clicking on the **Extension** | **Applications** link. Here, we can see the available applications that can be installed in our context:

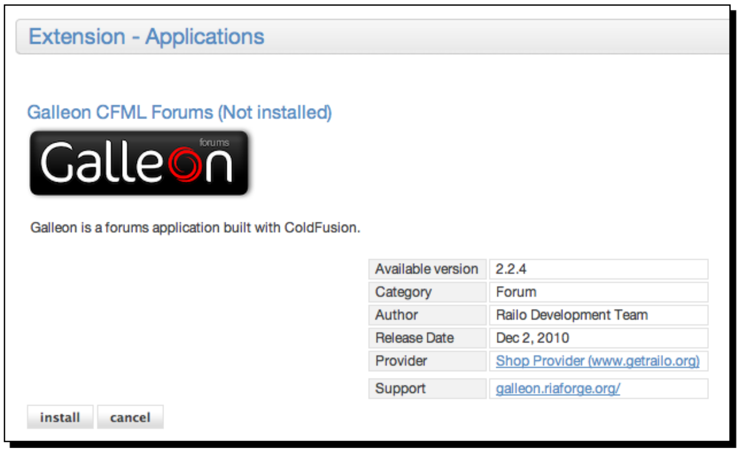


Extension - Applications

Not installed

These applications are not yet installed on the system.

3. Let's choose the **Galleon CFML Forums** and click on the option to see some more information about it:



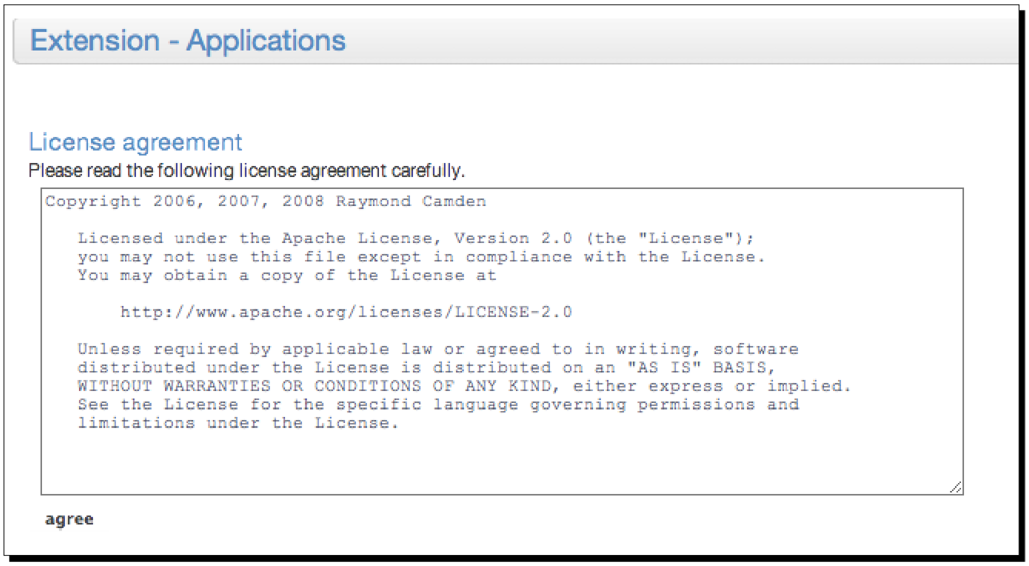
Extension - Applications

Galleon CFML Forums (Not installed)

Galleon is a forums application built with ColdFusion.

Available version	2.2.4
Category	Forum
Author	Railo Development Team
Release Date	Dec 2, 2010
Provider	Shop Provider (www.getrailo.org)
Support	galleon.riaforge.org/

4. Now that we are ready to install it, click on the **install** button and we are presented with a **License agreement** screen:



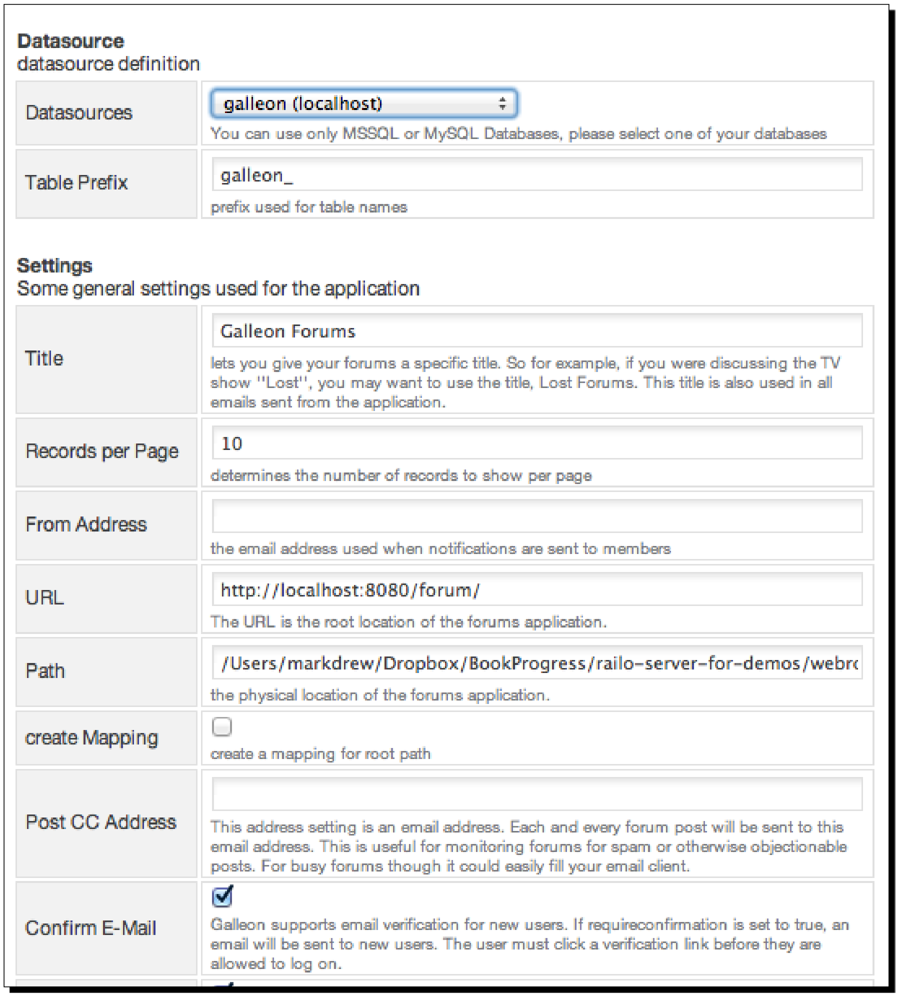
Extension - Applications

License agreement

Please read the following license agreement carefully.

```
Copyright 2006, 2007, 2008 Raymond Camden

   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.
```

5. Now, the next screen has a number of fields asking us for various details, such as which datasource we are going to use (let's select the Galleon datasource we created in the first step). It has a number of settings, but let's just add our e-mail address as that is all that is required to be added by us:



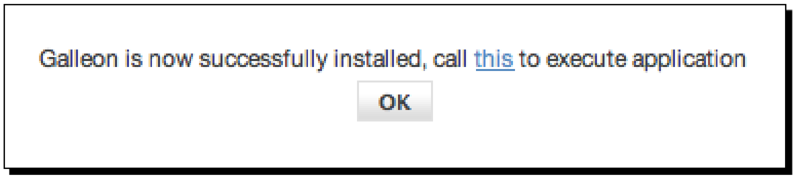
Datasource
datasource definition

Datasources	galleon (localhost) You can use only MSSQL or MySQL Databases, please select one of your databases
Table Prefix	galleon_ prefix used for table names

Settings
Some general settings used for the application

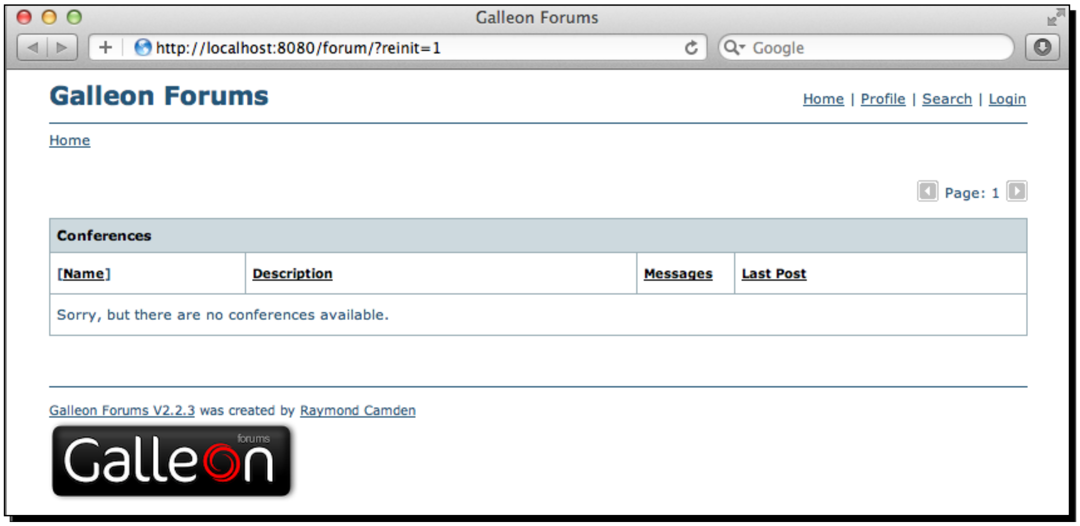
Title	Galleon Forums lets you give your forums a specific title. So for example, if you were discussing the TV show "Lost", you may want to use the title, Lost Forums. This title is also used in all emails sent from the application.
Records per Page	10 determines the number of records to show per page
From Address	the email address used when notifications are sent to members
URL	http://localhost:8080/forum/ The URL is the root location of the forums application.
Path	/Users/markdrew/Dropbox/BookProgress/railo-server-for-demos/webr the physical location of the forums application.
create Mapping	☐ create a mapping for root path
Post CC Address	This address setting is an email address. Each and every forum post will be sent to this email address. This is useful for monitoring forums for spam or otherwise objectionable posts. For busy forums though it could easily fill your email client.
Confirm E-Mail	☑ Galleon supports email verification for new users. If requireconfirmation is set to true, an email will be sent to new users. The user must click a verification link before they are allowed to log on.

6. Once we have filled everything, we can now click on the **install** button at the bottom and the application will be installed in our context.



Galleon is now successfully installed, call this to execute application

7. When we click on the link in this screen, we will be able to view the installed application:

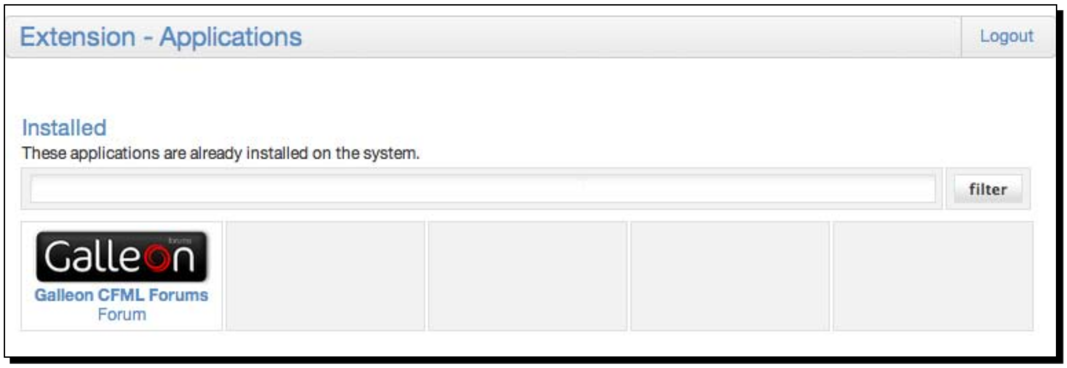

Galleon Forums
http://localhost:8080/forum/?reinit=1
Google
Home | Profile | Search | Login
Home
Page: 1

Galleon Forums

Conferences			
[Name]	Description	Messages	Last Post
Sorry, but there are no conferences available.			

Galleon Forums V2.2.3 was created by Raymond Camden

8 We are not going to go into how this application works here, but you can go ahead and try it out. When we click on the **OK** button on the installation confirmation screen, we will be taken back to the application's screen and we can see which applications we now have installed:



Extension - Applications

Installed

These applications are already installed on the system.

What just happened?

We installed the **Galleon CFML Forums** by using its Railo Server extension. We needed to provide some necessary information, such as our e-mail ID which was asked by the installer in the configuration screen. Then, we hit **install**, after which we installed Railo Server and configured our application for us.

We didn't need to download any external files, Railo Server did the whole installation for us. The extension installed the relevant database tables for us, and copied all the files to the correct location. Now, you can see what the benefit is by creating an extension for your application. It makes it extremely easy to package applications for other people to use and install.

Time for action – creating our own Railo application extension

Now that we know how to install extensions, it is about time to create our own.

An extension is a ZIP file that contains at least three files:

1. `license.txt` contains the license for the extension
2. `Install.cfc` handles the complete installation
3. `config.xml` contains all information about displaying the installation steps: field names, values, labels, page title, and so on

Creating the Famous Quotes App

For this exercise, we will create an application which we can then install. So let's create a Famous Quotes App:

- Under the `<Railo Install Directory>/webroot`, create a folder named `famousquotes`.
- Create a file `quotes.txt` in our `famousquotes` folder with the following content:

```
Houston, we've got a problem...
Oh my god, they killed Kenny!
… and any others you can come up with, divided by a line break
```

- Create `Quote.cfc` with the following content. This returns a random single line from the `quotes.txt` file:

```
<cfcomponent>
  <cffunction name="getquote" returntype="string" output="false">
    <cfset var filecontents = fileRead("quotes.txt") />
    <cfset var numberofquotes = listLen(filecontents, chr(10)) />
    <cfreturn listGetAt(fileContents, randRange(1,
numberofquotes), chr(10)) />
  </cffunction>
</cfcomponent>
```

- Create `include_quote.cfm` with the following content. This will call the `Quote.cfc` component and call our `getQuote()` function:

  ```
  <h3>Random quote</h3>
  <cfset quoteObj = createObject("component", "Quote") />
  <cfoutput><p><em>#quoteObj.getQuote()#</em></p></cfoutput>
  ```

- Create `QuoteWebservice.cfc`:

  ```
  <cfcomponent extends="Quote">
    <cffunction name="getquote" access="remote" returntype="string"
  output="false">
    <cfreturn super.getquote() />
    </cffunction>
  ```

  Create a ZIP file with these four files and name it `famousquotesapp.zipc`.

Alright, we got a killer app with included web service functionality and everything. It's time to make a Railo Server extension for it.

First, we need to think about how we want the application to be installed. Just think of a regular "next-next-finish" installer. In this case, we can just copy four files to a directory within the webroot. Also, we should ask if they actually want the web service to be installed as well, as it could be perceived as an undesired access point.

- We've got two questions we want to ask, which we will split into two installation steps. Create the `config.xml` file, with the following content:

  ```
  <?xml version="1.0"?>
  <config>
  </config>
  ```

- This file will be read by Railo Server to create our steps; let's add the first step that asks where you want to install the application. In the `config.xml` file, add the following:

  ```
  <?xml version="1.0"?>
  <config>
    <step label="Step 1" description="Thanks for installing the
  Famous Quotes app! &lt;br&gt; The following steps will guide you
  through installing the app.">
      <group label="Installation directory" description="Please
  enter the directory path starting from your webroot, where the
  Famous Quotes app will be installed into">
        <item type="text" name="installdir" label="Directory name or
  path">/quotesapp/</item>
      </group>
    </step>
  </config>
  ```

- Great! This gets where we want to install the application. Now let's add a second step, asking whether the user wants to install the webservice:

```
<?xml version="1.0"?>
<config>
  <step label="Step 1" description="Thanks for installing the
Famous Quotes app! &lt;br&gt; The following steps will guide you
through installing the app.">
    <group label="Installation directory" description="Please
enter the directory path starting from your webroot, where the
Famous Quotes app will be installed into">
      <item type="text" name="installdir" label="Directory name or
path">/quotesapp/</item>
    </group>
  </step>
  <step label="Step 2" description="Webservice installation">
    <group label="Install webservice?" description="Do you want
to install the webservice component as well? This will make it
possible for everyone to retrieve the quotes from your website.">
     <item type="radio" name="installWS" description="">
        <option value="1" description="">Yes</option>
      </item>
      <item type="radio" name="installWS" description="">
        <option value="0" description="">No</option>
      </item>
    </group>
  </step>
</config>
```

- We need to validate the input, copy files, and have a function to uninstall. Let's create the `Install.cfc` file for this task:

```
<cfcomponent output="no">

  <cffunction name="validate" returntype="void" output="no">
    <cfargument name="error" type="struct" required="yes" />
    <cfargument name="path" type="string" required="yes" />
    <cfargument name="config" type="struct" required="yes" />
    <cfargument name="step" type="num+++++
++
eric" required="yes" />
    <cfset var allformdata = config.mixed />
    <!--- the install directory --->
    <cfif arguments.step eq 1>
      <cfif left(allformdata.installdir, 1) neq "/" or
right(allformdata.installdir, 1) neq "/">
```

```
        <cfset arguments.error.fields.installdir = "The directory
must both start and end with a forward slash (i.e. /foldername/)"
/>
      </cfif>
      <!--- Does a file "quotes.txt" exist in the directory? If
so, we don't allow overwrite. --->
      <cfif fileExists(expandPath(allformdata.installdir &
"quotes.txt"))>
        <cfset arguments.error.fields.installdir = "The directory
[#allformdata.installdir#] already exists, and contains a
file [quotes.txt]. Please backup and remove this file before
continuing." />
      </cfif>
    <cfelse>
      <cfif not structKeyExists(allformdata, "installWS")>
        <cfset arguments.error.fields.webservertype = "Please
choose if you want to install the webserver file." />
      </cfif>
    </cfif>
  </cffunction>
</cfcomponent>
```

- The `Install.cfc` now has one method, named `validate`; this method will check whether the user has entered the proper values for the location, whether we are overwriting any existing files, and if they have chosen to install the web service.
- Next, let's add the `install` method to our `Install.cfc` file:

```
<cfcomponent output="no">

  <cffunction name="validate" returntype="void" output="no">
    ...
  </cffunction>

  <cffunction name="install" returntype="string" output="no">
    <cfargument name="error" type="struct" required="yes" />
    <cfargument name="path" type="string" required="yes" />
    <cfargument name="config" type="struct" required="yes" />
    <cfset var allformdata = arguments.config.mixed />
    <cfset var sReturn = "" />
    <cfset var savePath = expandPath(allformdata.installdir) />
    <!---  create the destination directory when necessary --->
    <cfif not directoryExists(savepath)>
      <cfdirectory action="create" directory="#savepath#"
recurse="yes" />
    </cfif>
    <!---  copy the app files to the right location --->
    <cfzip action="unzip" file="#arguments.path#famousquotesapp.
zip" destination="#savepath#" filter="*.*" overwrite="yes" />

    <!---  remove the webservice file, if requested --->
    <cfif allformdata.installWS neq 1>
      <cffile action="delete" file="#savepath#QuoteWebservice.cfc"
/>
    </cfif>
    <!--- give a response --->
    <cfsavecontent variable="sReturn"><cfoutput>
      <h3>The <em>Famous Quotes app</em> has been installed!</h3>
      <p><a href="#allformdata.installdir#include_quote.cfm">See
the app in action</a></p>
      <cfif allformdata.installWS neq 1>
        <p><em>Note: the webservice files were not installed.</
em></p>
      </cfif>
      <p></p>
    </cfoutput></cfsavecontent>
    <cfreturn sReturn />
  </cffunction>
</cfcomponent>
```

- The `install` method takes care of copying the right files, including checking whether we need to also remove the web service. It does this by using the `famousquotesapp.zip` file we created previously and unpacking the files.
- Now that we have installed our extension, let's add the functionality to uninstall it. Add the `uninstall` method to our `Install.cfc` component:

```
<cfcomponent output="no">

  <cffunction name="validate" returntype="void" output="no">
    ...
  </cffunction>

  <cffunction name="install" returntype="string" output="no">
    ...
  </cffunction>

  <cffunction name="uninstall" returntype="string" output="no">
    <cfargument name="path" type="string" required="yes" />
    <cfargument name="config" type="struct" required="yes" />
    <cfset var allformdata = arguments.config.mixed />
    <cfset var savePath = expandPath(allformdata.installdir) />
    <cfset var qFiles = "" />
    <cfset var aErrors = [] />

    <!--- find out which files we installed, so we can remove
those --->
    <cfzip action="list" name="qFiles" file="#arguments.
path#famousquotesapp.zip" filter="*.*" />
    <!---  keep the quotes.txt file; it might contain user-
provided content --->
    <cfloop query="qFiles">
      <cfif qFiles.name neq "quotes.txt">
        <cftry>
          <cffile action="delete" file="#savePath##qFiles.name#"
/>
          <cfcatch>
            <cfset arrayAppend(aErrors, "The file
#savePath##qFiles.name# could not be deleted: #cfcatch.message#
#cfcatch.detail#.") />
          </cfcatch>
        </cftry>
      </cfif>
    </cfloop>

    <cfset var sReturn = "" />
    <cfsavecontent variable="sReturn"><cfoutput>
      <p><strong>The <em>Famous Quotes App</em> is now
uninstalled.</strong><p/>
      <p>To prevent accidental loss of your data, the file
<em>quotes.txt</em> has not been removed.</p>
      <!---  errors occurred? Show them here --->
      <cfif not arrayIsEmpty(aErrors)>
        <p style="color:red;">One or more errors occurred while
uninstalling:</p>
        <ul style="color:red;">
          <cfset var i = 0 />
          <cfloop collection="#aErrors#" item="i">
            <li>#aErrors[i]#</li>
          </cfloop>
        </ul>
      </cfif>
    </cfoutput></cfsavecontent>

    <cfreturn sReturn />
  </cffunction>

</cfcomponent>
```

- The uninstall method gets a list of files from our famousquotesapp.zip file and deletes these files from the folder in which the application was originally installed in. It doesn't remove the quotes file as, of course, the user might have changed it.
- Finally, let's use a final method; this time, it's the update method, which will be run if there is a new version of the extension:

  ```
  <cfcomponent output="no">

    <cffunction name="validate" returntype="void" output="no">
      ...
    </cffunction>

    <cffunction name="install" returntype="string" output="no">
      ...
    </cffunction>

    <cffunction name="uninstall" returntype="string" output="no">
      ...
    </cffunction>

    <cffunction name="update" returntype="string" output="no">
      <cfreturn install(argumentCollection=arguments) />
    </cffunction>
  </cfcomponent>
  ```

- The last thing we need to do is create a license.txt file. Here, we are using the Apache 2 license:

  ```
  Copyright 2011 <Your_name>

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this software 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.
  ```

- Now that we've got all the files needed to create our own extension, we need to make a ZIP file containing all the documents we just created. This ZIP file will be our Railo Extension.
- Create a ZIP file, named `famousquotesinstaller.zip`, with the following files:
    - `famousquotesapp.zip`
    - `config.xml`
    - `Install.cfc`
    - `license.txt`

What just happened?

You are probably kind of disappointed at this point, because you cannot see the result of all the hard work you just did. I am sorry for that, and I will make it up to you in the upcoming pages.

So, what just happened? We created a first-class *Famous Quotes App*, and we made a Railo extension for it, which we will absolutely be installing—just hang on for a few paragraphs.

We created a `config.xml` file, which contains the *steps* (HTML pages) we need to go through for installing the extension. It describes the HTML frontend of the installation procedure. Each *step* within the XML contains one or more *groups*, which contains one or more *item* tags. These *items* will be shown as HTML form elements during installation, such as `<input type="text">`, `<input type="radio">`, and `<select>`.

> For more in-depth knowledge of the configuration options, see: `http://wiki.getrailo.org/wiki/Tutorial:Extension_Provider`.

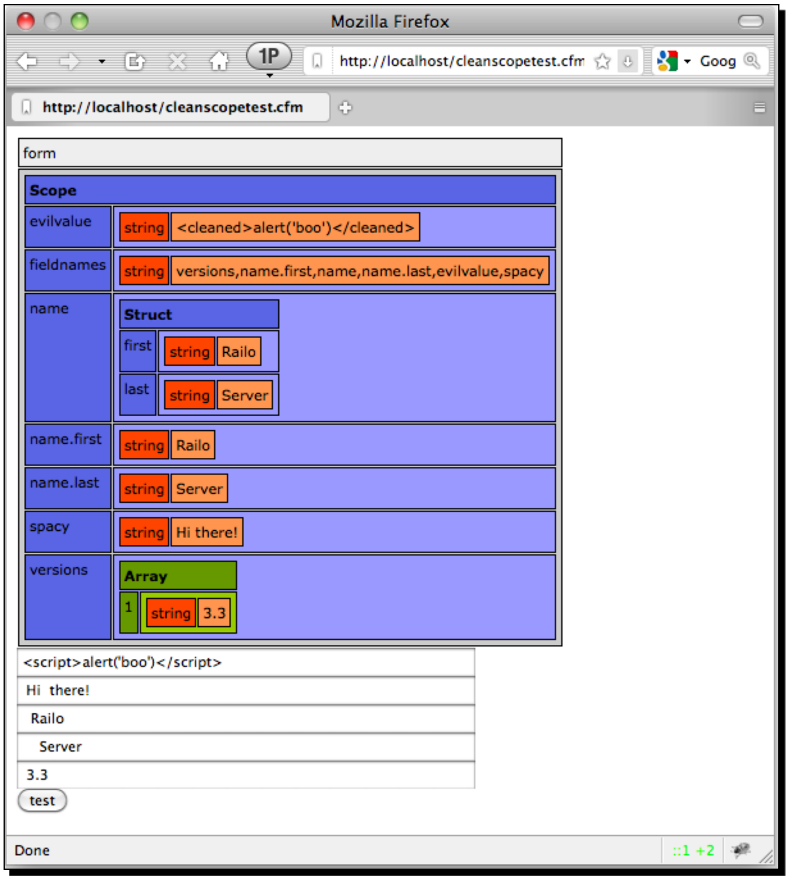
Then, we created a fairly long `Install.cfc` component, which handles the complete installation, updating, and uninstallation of our extension.

After each of the steps defined in the XML file, the `validate` function within `Install.cfc` is called. This function checks the entered data. If you want to prevent the user from going to the next step because a problem is found with the given data, you can just add an error to the given error structure in the `arguments` scope:

```
<cfif not structKeyExists(allformdata, "installWS")>
  <cfset arguments.error.fields.webservertype = "Please choose if you want
to install the webserver file." />
</cfif>
```

If the `arguments.error` structure remains empty, then the installation goes to the next step.

After all validation steps, the `install` function is called. We receive all submitted data in the `config` argument and the absolute path to our installation package in the `path` argument. When doing a `<cfdump eval=arguments />`, you will see the following:


Mozilla Firefox
1P
http://localhost/cleanscopetest.cfm
Goog
http://localhost/cleanscopetest.cfm
Done
:1 +2

```
form
Scope
evilvalue    string  <cleaned>alert('boo')</cleaned>
fieldnames   string  versions,name.first,name,name.last,evilvalue,spacy
name         Struct
             first   string  Railo
             last    string  Server
name.first   string  Railo
name.last    string  Server
spacy        string  Hi there!
versions     Array
             1   string  3.3
```

```
<script>alert('boo')</script>
Hi  there!
 Railo
   Server
3.3
test
```

`Install.cfc` simply copies our *Famous Quotes App* files to the requested directory by unzipping the zipped application:

```
<cfzip action="unzip" file="#arguments.path#famousquotesapp.zip"
destination="#savepath#" filter="*.*" overwrite="yes" />
```

Then, all we need to do is return an **installation successful** message.

We also needed to provide an `uninstall` function. This function is called from the Railo Administrator when a user chooses to uninstall the application. This function receives the same argument values as the `install` function received while installing. To make this possible, all installation variables are automatically written to the configuration XML of the Railo administrator (`railo-server.xml` / `railo-web.xml.cfm`).

Time for action – creating our own extension provider

We created the Railo Extension `famousquotesinstaller.zip`, which we now have on our local computer. How can we add and share the extension within Railo Server? This can be achieved by using the **Extension Provider system**. This is a system unique to Railo Server, where you can easily share extensions with other Railo Server users.

We only need to create one CFC file, which will be serving our extensions for us, and provides some details about itself.

1. At the root of our `<Railo Install Directory>/webroot2/` folder, create a file named `ExtensionProvider.cfc`, with the following content:

```
<cfcomponent output="no">

  <cffunction name="getInfo" access="remote" returntype="struct"
output="no">
    <cfset var info = {
        title="Railo book extensions"
      , description="Providing you with the best Extensions you
can find!"
      , image="http://site2.local:8888/railoBookExtensions.png"
      , url="http://site2.local/"
      , mode="develop"
    } />
    <cfreturn info />
  </cffunction>

</cfcomponent>
```

2. The previous code adds the `getInfo` method to our extension provider, which returns some information about the extension provider itself. Next, let's add the `listApplications` method, which will obviously list our applications:

```
<cfcomponent output="no">

  <cffunction name="getInfo" access="remote" returntype="struct"
output="no">
    …
  </cffunction>

  <cffunction name="listApplications" access="remote"
returntype="query" output="no">
    <cfset var apps = queryNew("type,id,name,label,description,ver
sion,category,image,download,author,codename," &
```

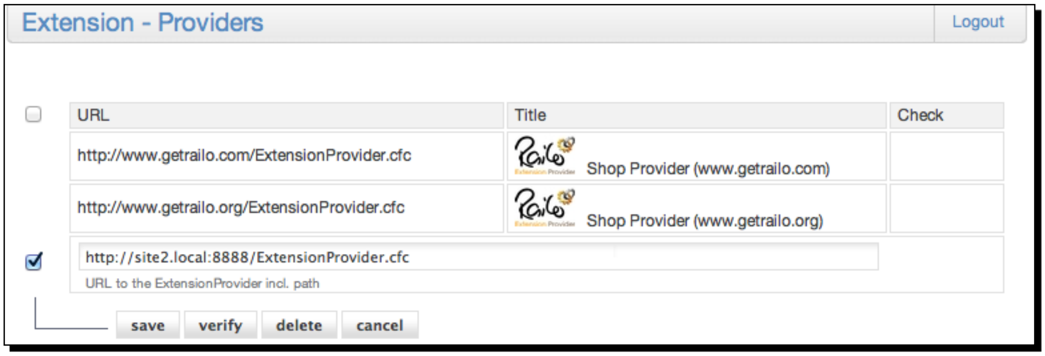
```
      "video,support,documentation,forum,mailinglist,network,creat
ed") />

    <cfset QueryAddRow(apps) />
    <cfset QuerySetCell(apps, "id", "famousquotes") />
    <cfset QuerySetCell(apps, "name", "Famous Quotes App") />
    <cfset QuerySetCell(apps, "type", "web") />
    <cfset QuerySetCell(apps, "label", "Famous Quotes App") />
    <cfset QuerySetCell(apps, "description",  "This extension will
install the Famous Quotes App") />
    <cfset QuerySetCell(apps, "author", "[Your name]") />
    <cfset QuerySetCell(apps, "codename", "railobookV1") />
    <cfset QuerySetCell(apps, "support", "http://site2.local/
support/") />
    <cfset QuerySetCell(apps, "created", CreateDate(2011, 06, 01))
/>
    <cfset QuerySetCell(apps, "version", "1.0.0") />
    <cfset QuerySetCell(apps, "category", "Applications") />
    <!--- if you do NOT define a URL here, the function
getDownloadDetails() is called --->
    <cfset QuerySetCell(apps, "download", "") />

    <cfreturn apps />
  </cffunction>


</cfcomponent>
```

3. In the previous code, we create a query object and add a row to it, adding information to all the columns about our *Famous Quotes Apps*. Finally, let's add the `getDownloadDetails` method, which will be called when someone installs our application:

```
<cfcomponent output="no">

  <cffunction name="getInfo" access="remote" returntype="struct"
output="no">
  ...
  </cffunction>

  <cffunction name="listApplications" access="remote"
returntype="query" output="no">
  ...
  </cffunction>
  <cffunction name="getDownloadDetails" access="remote"
returntype="struct" output="no">
    <cfargument name="type" required="yes" type="string" />
    <cfargument name="serverId" required="yes" type="string" />
    <cfargument name="webId" required="yes" type="string" />
    <cfargument name="appId" required="yes" type="string" />
    <cfargument name="serialNumber" required="no" type="string"
default="" />

    <cfset var linebreak = server.separator.line />
    <cflog file="extensiondownloads" type="information" text="Date
:#now()##linebreak#IP:#cgi.REMOTE_ADDR##linebreak#arguments:#seria
lize(arguments)#" />

    <cfset var data = structNew() />
    <cfif arguments.appId eq "famousquotes">
      <cfset data.error = 0 />
      <cfset data.url = "http://#cgi.http_host#/
famousquotesinstaller.zip" />
    <cfelse>
      <cfset data.error = 1 />
      <cfset data.url = "" />
      <cfset data.message = "The extension you requested does not
exist." />
    </cfif>

    <cfreturn data />
  </cffunction>
</cfcomponent>
```

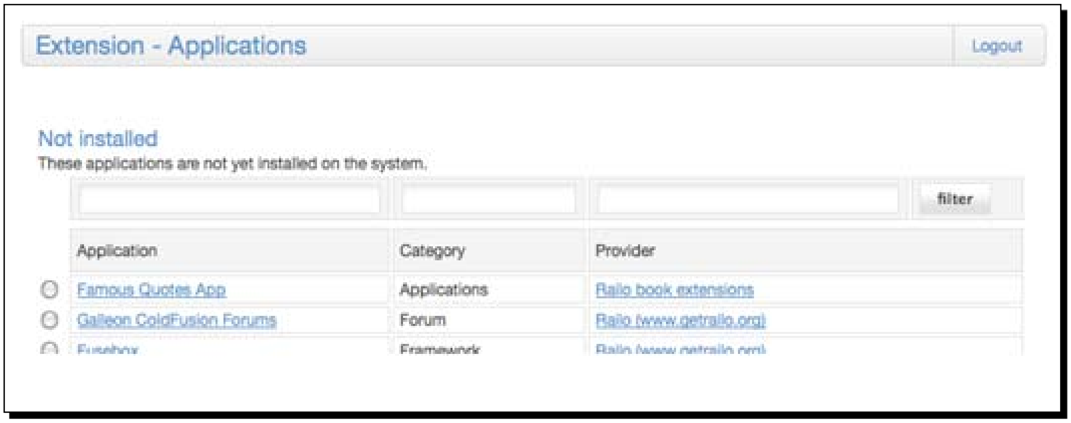
4. This method checks if the extension exists and returns the url of the `famousquotesinstaller.zip` file

5 Now go to the Railo web admin at `http://localhost:8888/railo-context/admin/web.cfm` and navigate to **Extensions** | **Providers**. Check the checkbox, and enter our extension provider's URL `http://site2.local:8888/ExtensionProvider.cfc` in the textbox next to it:



□	URL	Title	Check
	http://www.getrailo.com/ExtensionProvider.cfc	Shop Provider (www.getrailo.com)	
	http://www.getrailo.org/ExtensionProvider.cfc	Shop Provider (www.getrailo.org)	
☑	http://site2.local:8888/ExtensionProvider.cfc URL to the ExtensionProvider incl. path		

6. After we hit **submit**, we can navigate to **Extensions** | **Applications** within the administrator. There, we can finally see our *Famous Quotes App* listed:



Not installed

These applications are not yet installed on the system.

	Application	Category	Provider
○	Famous Quotes App	Applications	Railo book extensions
○	Galleon ColdFusion Forums	Forum	Railo (www.getrailo.org)
○	Fusebox	Framework	Railo (www.getrailo.org)

What just happened?

We created a CFC file, which provides information about our extensions. Most of the functions in the CFC have their `access` attribute set to `remote`, which means they can be called as a web service:

```
<cffunction name="getInfo" access="remote" ...>
```

We saved it in the `webroot` of `site2.local`, so we can access it from the URL `http://site2.local:8888/ExtensionProvider.cfc`.

Then, we added this URL into the Railo Web Administrator as a new Extension provider. After doing this, we saw our *Famous Quotes App* appear in the **Applications** list of the administrator.

Let's go into some more detail about the CFC we just created.

The ExtensionProvider CFC

First off, you can name the file anything you want. It could even be a .NET web service if you wanted it to; but then you wouldn't be reading this book now, would you?

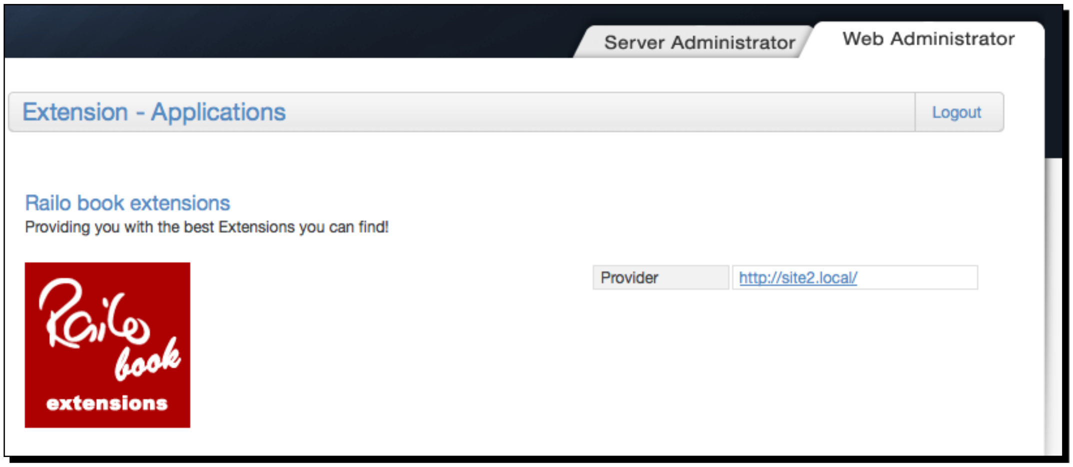
Anyway, the CFC has a few required methods:

- `getInfo`: This method returns a structure with information about your Extension Provider.
- `listApplications`: This one returns a query with all the available extensions.
- `getDownloadDetails`: This returns a structure with download information for a given extension. Only used/required when no download URL is given in the extension details query.

GetInfo structure information

Key	**Description**
title	Title of the extension provider
description	Description of the extension provider
image	Link to an image
URL	URL for more information
mode	Defines how the information of the `ExtensionProvider` is cached in the Railo Administrator. The valid values are: • `develop`—does not cache the result of the extension provider • `production` (`or no value`)—caches the result in the session scope of the Railo administrator user

This information is partly shown in the **Details** page of an extension, and fully when you click on the Provider's name in the **Applications** listing:



ListApplications query information

As we have seen in the previous example, we needed to create a query with information about our extension. The following list is very long, but most of the fields can remain empty. The required fields are marked with an asterisk (*).

Field	Description
Id *	This is the identifier of the extension. With this ID, updates are done and download details are retrieved.
Name *	This is the internal name of the extension. It must be a valid CFML variable name.
Label *	This displays the name of the extension.
Description *	This is the text/HTML description of the extension.
Type	This determines if the extension can be installed via the Web and/or Server Administrator. The valid values are: server, web, and so on. The value "web" is the default value.
Download	It is the URL to the extension itself. If empty, the function `getDownloadDetails()` is called when a user wants to install the extension.
Author	Author of the extension.
Created	The date of when was the extension created.
Version *	Version number of the extension.
Codename	Code name for (the current release of) the extension.
Category	Category of the extension. For example "CMS", "Framework", "Database".
Video	URL to a video file, for example, for installation instructions.
Image	URL to an image file, which will be used as the logo for the extension.

Field	Description
Support	Support URL for the extension
Documentation	URL to documentation about the extension
Forum	URL to a forum about the extension
Mailinglist	URL to a mailing list about the extension
Network	URL to a users' network about the extension

GetDownloadDetails function

As said before, this function returns the download details about a given extension. There are a lot of benefits when using this function, instead of just adding the download link to the query of `listApplications()`. The most important one is that you can track how many people downloaded your extension, as we saw in the code we created:

```
<cfset var linebreak = server.separator.line />

<cflog file="extensiondownloads" type="information" text="Date:#now()
##linebreak#IP:#cgi.REMOTE_ADDR##linebreak#arguments:#serialize(argum
ents)#" />
```

Because the function is only called when a Railo Administrator user chooses to install the plugin, you can be fairly certain that the download count is the sum of all installations and updates.

Another good reason to use this function is for payment handling. If we want to become rich with our *Famous Quotes App*, we could add a simple check like this:

```
<cfif not hasPaidForExtension(arguments.appID, arguments.webID)>
  <cfset data.error = 1 />
  <cfset data.url = "" />
  <cfset data.message = "Please first pay $25 to our paypal account
paypal@famousq.com. Afterwards, email us your server details by <a
href='mailto:sales@famousq.com?body=#urlencodedformat('$25 was paid to
the paypal account, for web ID #arguments.webID#')#'>clicking here</
a>" />
<cfelse>
  [ ... send them the download details ... ]
```

The structure returned by this function must contain the fields, as shown in the following table:

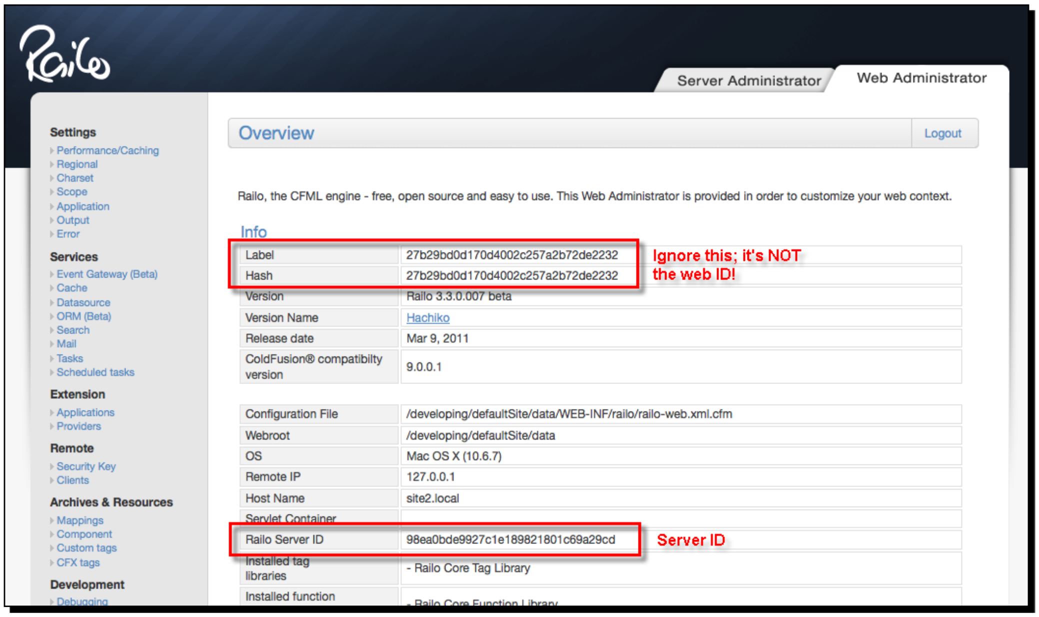
Key	Description
Error	A number indicating whether an error should be shown: ◆ 0 : No error. The "URL" value must be given. ◆ 1 : An error occurred. The "message" field contains the error message.
URL	The URL to the extension
Message	The error message

The role of the Web ID and Server ID

You probably noticed that we receive a `webID` and `serverID` as arguments for the `getDownloadDetails` function. These IDs can uniquely identify any Railo Server and web context.

The **Server ID** is created by Railo at the moment Railo Server is started for the first time. The **Web ID** is created at the moment a web context is first called. Both IDs will never change and are stored in the Railo configuration files.

You can view the Server ID on the **Dashboard** page of the Railo Administrator. Even though there is another ID shown on the dashboard page, which is labelled "Hash", this is *not* the Web ID:



Railo, the CFML engine - free, open source and easy to use. This Web Administrator is provided in order to customize your web context.

Info

Label	27b29bd0d170d4002c257a2b72de2232
Hash	27b29bd0d170d4002c257a2b72de2232
Version	Railo 3.3.0.007 beta
Version Name	Hachiko
Release date	Mar 9, 2011
ColdFusion® compatibilty version	9.0.0.1
Configuration File	/developing/defaultSite/data/WEB-INF/railo/railo-web.xml.cfm
Webroot	/developing/defaultSite/data
OS	Mac OS X (10.6.7)
Remote IP	127.0.0.1
Host Name	site2.local
Servlet Container	
Railo Server ID	98ea0bde9927c1e189821801c69a29cd
Installed tag libraries	- Railo Core Tag Library
Installed function	- Railo Core Function Library

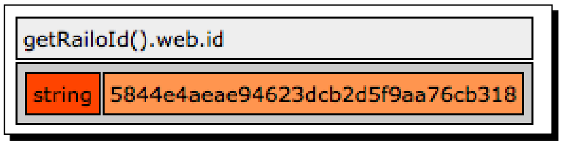
Ignore this; it's NOT the web ID!

Server ID

The Web ID can be found programmatically by running the following code from somewhere in the Web context:

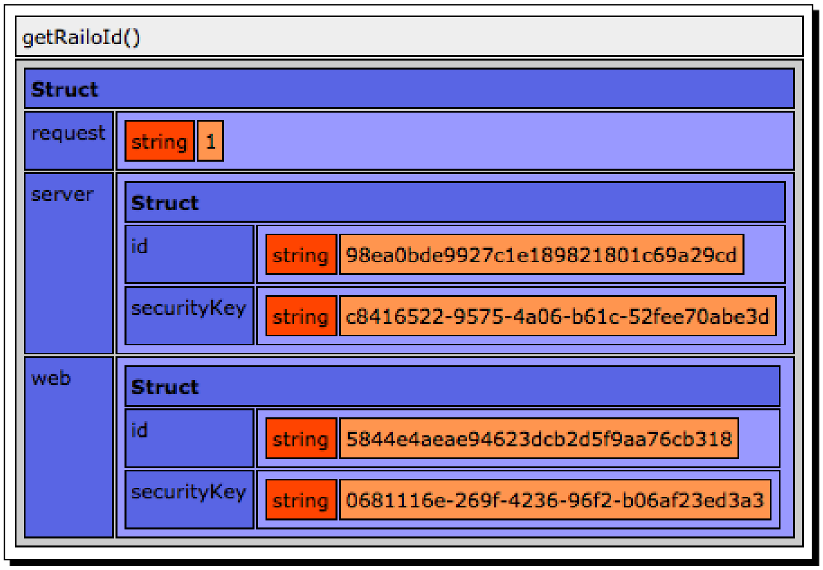
```
<cfdump eval=getRailoId().web.id />
```

This will output the following:



```
getRailoId().web.id
string  5844e4aeae94623dcb2d5f9aa76cb318
```

If you think like me, then the question currently in your head is, "So what does the complete `getRailoID()` output look like?". Well, let's not keep you waiting for an answer. It looks as follows:



```
getRailoId()
Struct
request   string  1
server    Struct
          id           string  98ea0bde9927c1e189821801c69a29cd
          securityKey  string  c8416522-9575-4a06-b61c-52fee70abe3d
web       Struct
          id           string  5844e4aeae94623dcb2d5f9aa76cb318
          securityKey  string  0681116e-269f-4236-96f2-b06af23ed3a3
```

As you can see, it also contains the Web ID, the Server ID, and a **securityKey**. This security key has nothing to do with the extensions, but instead belongs to the **Admin Synchronization** system.

> The **Admin Synchronization** system lets you manage multiple Railo administrators from one place. Any change you make in one place is instantly sent to all linked administrators. For more information on this paid extension see `http://www.railo.ch/blog/index.cfm/2008/9/10/Railo-30-released--Features-part-2`.

Let's get back on topic to the Web and Server IDs. When we installed our extension, we had to get the download details from the function `getDownloadDetails()`. We sent the Web ID of our current web context (local host), and the Server ID of our Railo instance. Our extension provider can make decisions based on these IDs. Has the client paid for the extension yet? And if we log the IDs to a database, we will also know how many times this extension has been installed on a server and whether our clients already downloaded the latest update. If they previously registered with us because they paid for an extension, we can even send them an e-mail with instructions, or just to say thanks, the moment they download the extension.

Extend your ExtensionProvider

I hope you have learned enough to be able to extend the `ExtensionProvider.cfc` yourself. We already saw a very simple example of payment handling within the provider, but can you make this more robust?

Here are some things that could be added:

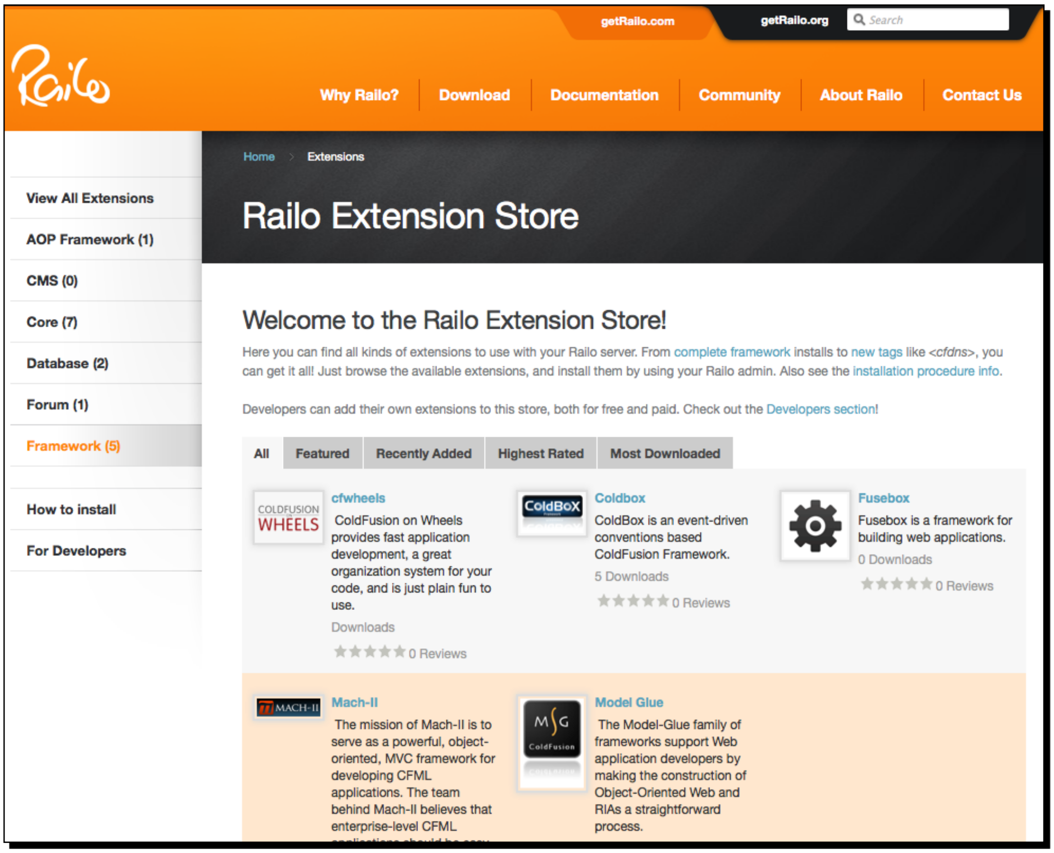
- Database schema for customers, extensions, downloads, and payments
- Saving download information to the database
- Checking in the database if a payment was made for the given server/web context
- The payment system itself

The Railo Extension Store

Railo 3.3 has a great new feature, which makes sure that any extension you create gets the attention it deserves. Like Apple has its App Store, Railo now has an Extension Store.

The store can be found at `http://www.getrailo.org/index.cfm/extensions/` and all its extensions are shown within your Railo Administrator.

You can browse through all the available extensions, register for a developer account, and add your own extensions:



Railo Extension Store

Welcome to the Railo Extension Store!

Here you can find all kinds of extensions to use with your Railo server. From complete framework installs to new tags like *<cfdns>*, you can get it all! Just browse the available extensions, and install them by using your Railo admin. Also see the installation procedure info.

Developers can add their own extensions to this store, both for free and paid. Check out the Developers section!

cfwheels
ColdFusion on Wheels provides fast application development, a great organization system for your code, and is just plain fun to use.
Downloads
0 Reviews

Coldbox
ColdBox is an event-driven conventions based ColdFusion Framework.
5 Downloads
0 Reviews

Fusebox
Fusebox is a framework for building web applications.
0 Downloads
0 Reviews

Mach-II
The mission of Mach-II is to serve as a powerful, object-oriented, MVC framework for developing CFML applications. The team behind Mach-II believes that enterprise-level CFML

Model Glue
The Model-Glue family of frameworks support Web application developers by making the construction of Object-Oriented Web and RIAs a straightforward process.

It is your choice if you want to use the **Railo Extension Store** or your own provider. Here are the main differences between the two:

Option	Extension Store	Own provider
Paid extensions	You can't add paid extensions to the store (there is a **Contact us** link though)	You can build your own payment handling
Exposure	The Store Extensions are shown in every Railo Administrator worldwide	You'll have to tell people to add your provider to the Railo Administrator
Statistics	The store only has one statistic—the total number of downloads	You can log every request, which gives you more insight into its use
Hosting	Done by the Extension Store	Done by yourself

Summary

We learned about extending Railo Server in this chapter, by using the Extension system, custom tags, and custom functions.

Specifically, we covered:

- Creating and installing a custom tag in Railo Server. We learned how to enhance the `<cfabort>` tag, and where to save it so it will be picked up as a custom tag.
- Creating a custom function: `cleanScope`. We also saw how to use array and `struct` notation with form variables.
- How to use the Railo Administrator to install extensions.
- How to create our own Railo Extension. We created a tiny little web application and then created and zipped the necessary installation files.
- By creating and exploring `ExtensionProvider.cfc`, we learned how we can distribute our own extension. We also logged downloads and talked a bit about hosting paid extensions.
- After all the trouble of creating the `ExtensionProvider`, we found out that there is an extension store where we can simply upload our extension into. We also did a small comparison between the Store and a self-hosted extension provider.

Now that we have learned about extending Railo Server, let's build a full application. In the next chapter, we are going to build a video-sharing application from start to finish.

Let's get coding!

10

Creating a Video-sharing Application

In the preceding chapters, we have looked at a lot of features that are available to you in Railo Server. It's time to put them all into practice. This chapter will guide you through developing a video sharing application from start to finish. Also, in this process, we are going to put into practice techniques we learned in previous chapters so you can see how the various aspects of Railo Server actually work in a real world example.

In this chapter, we will:

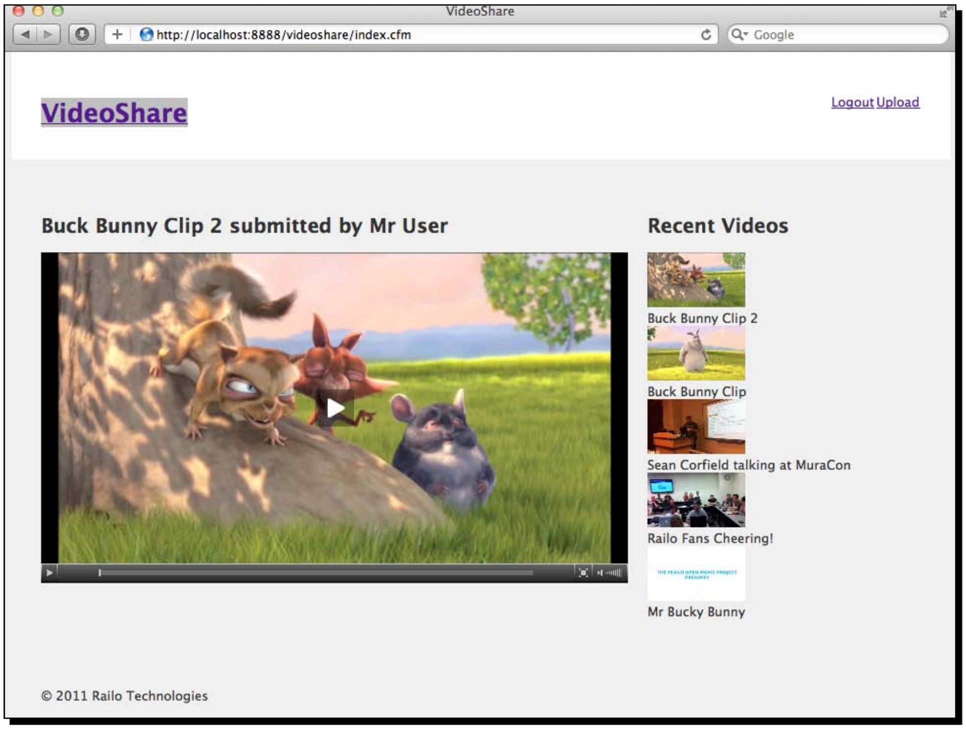
- Build an application to share videos
- Use the Railo Server ORM system to register and log in users
- Use the video conversion capabilities to convert videos ready for the web
- Use the session scope to allow users to log in to the application

Doesn't sound like a lot, but with these aspects, we can build a fully working application. Let's get to it!

VideoShare: Getting to know our application

Before we get coding, let's get acquainted with the application we are going to be building. VideoShare is a video-sharing website. It allows users to upload a video, which is then converted into a format suitable for display on the Web. It also allows registered users to comment on those videos.

When finished, our application should look like the following screenshot:


VideoShare
http://localhost:8888/videoshare/index.cfm
Google
VideoShare
Logout Upload
Buck Bunny Clip 2 submitted by Mr User
Recent Videos
Buck Bunny Clip 2
Buck Bunny Clip
Sean Corfield talking at MuraCon
Railo Fans Cheering!
Mr Bucky Bunny
© 2011 Railo Technologies

Goals of the application

Before we get coding, let's make a list of the things that the video sharing application should do:

- Be able to show a home page with a main video and a list of all the other items that have been recently uploaded
- Be able to register a user so that they can upload a video
- Allow users to log in and log out of the application
- Only allow registered users to upload a video
- Convert a video into a format that is web viewable
- Create a large poster image of the video
- Create a thumbnail version of the poster image

Other goals are:

- Allow registered users to comment on a video
- Display comments underneath a video

That seems to be enough to get on, so let's start implementing these features.

Creating our application

Let's get the ball rolling by creating our application skeleton and adding our initial settings.

Time for action – creating our basic application

To start off, let's just create a place to store our application.

1. In the `<Railo Install Location>/wwwroot/` path, let's create a folder named `VideoShare`. This might as well be the name of our application.

2. In the `VideoShare` folder, create a file named `Application.cfc` and add the following code:

```
component {
  this.name = "VideoShare";
  this.datasource = "VideoShare";
  this.ormEnabled = true;
  this.sessionmanagement = true;
  this.ormSettings = {};
  this.ormSettings.dbcreate = "dropCreate";
}
```

Let's look at the previous code for a second as there is a lot going on. We already know that the `Application.cfc` file controls the settings in an application; so by creating a new one here, we can manage various settings. The first setting is the name of the application, in this case, we have named it `VideoShare` and then we have defined that it will use a datasource named `VideoShare`, which we will set up in the next section. Because we will want to track user sessions to see if they are logged in or not, we are setting `this.sessionmannagement = true`. We shall look further into how we manage it later on.

Then comes the magic: we set `this.ormEnabled=true;` which enables us to define our data domain using only Railo Components and not have to bother with playing around with the database itself.

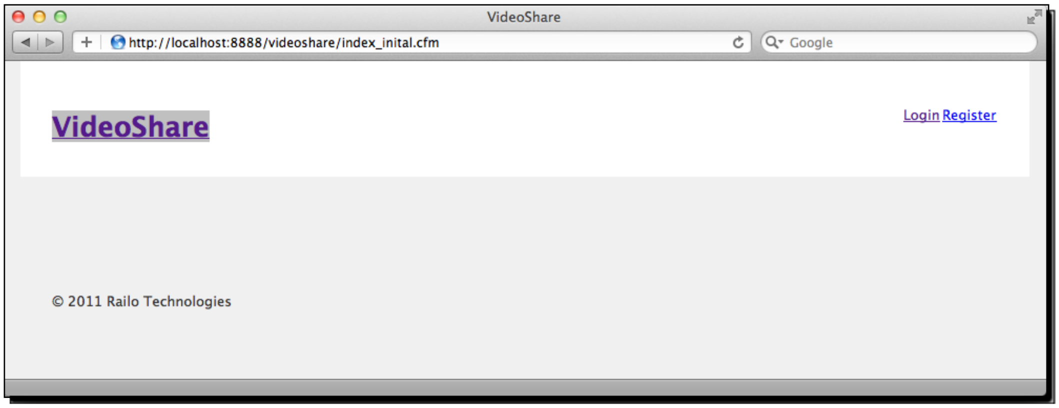
3. Because we are in development mode, we will be changing the schema (or structure) of our database, so we need to set some settings for the ORM by first creating the `this.ormSettings = {};` structure. To this structure, we add `this.ormSettings.dbcreate = "dropCreate"`, which tells Railo server that if we change our ORM-persisted components, Railo Server will drop and re-create the database tables. Be warned—this means that any data we have there will also be deleted, so it's only good while we are developing.

4. We have got all our settings in place, now let's get our first page set up. In the same folder that we created our `Application.cfc` file, let's create our `index.cfm` file and put the following content in it:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
  <link rel="stylesheet" href="css/main.css" type="text/css">
<title>VideoShare</title>
</head>
<body>
<cfoutput>
<div id="header">
    <h1><a href="index.cfm">VideoShare</a></h1>
    <div id="topnav">
      <ul>
        <li><a href="login.cfm">Login</a></li>
        <li><a href="register.cfm">Register</a></li>
      </ul>
    </div>
</div>
<div id="main">
  <div id="content">
    Welcome!
  </div>
</div>
<div id="footer">&copy; #DateFormat(Now(), "yyyy")# Railo
Technologies</div>
</cfoutput>
</body>
</html>
```

5. The previous code should give you an idea of the layout we are going to use. We have three main divs in our layout, namely, `header`, `main`, and `footer`.

6. You should have also noticed that we are using a stylesheet. Let's create a folder named `css` in our current working folder and copy the `main.css` from the source code files available for this book from the PacktPub site; they should be under `videoshare/css/main.css`. Once you have added these, hopefully your site will look as follows:



VideoShare

Login Register

© 2011 Railo Technologies

7. By now, you should be all set to actually start creating our application.

What just happened?

Before we dive into our application, let us have a look at the nice layout we have ready. We have set up our application with the ORM settings used while we were developing. To make the site look a bit better, we have added a stylesheet to the code.

Have a go hero – create a datasource

Because we have defined the name of the datasource as `VideoShare` in our `Application.cfc`, we need to create that now, even though we are not going to use it yet.

You have done this a number of times before in this book, so why don't you try creating a new database in MySQL and creating a datasource in the Railo Web Administrator? It should be pretty easy by now.

Laying it all out

There is a problem here with our initial strategy. Because we are going to be adding a few more pages and want to let them have the same look and feel, we could copy the `index.cfm` page each time, but this would mean that if we wanted to change the layout, or add any logic to the navigation, we would have to make the change to *every* page. Instead of doing that, we can use a custom tag to create our layout, and just call that from each page.

Time for action – creating the layout custom tag

We are going to split out the main part of our layout into a custom tag. A custom tag is handy for this, because we can call it from all the pages:

1. Let's first create a file in our `videoshare` directory named `layout.cfm`. Copy all the contents of `index.cfm` into this file, because so far, all we have is something we are going to repeat in our code. `layout.cfm` should now look like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
  <link rel="stylesheet" href="css/main.css" type="text/css">
  <title>VideoShare</title>
</head>
<body>
<cfoutput>
<div id="header">
    <h1><a href="index.cfm">VideoShare</a></h1>
    <div id="topnav">
      <ul>
        <li><a href="login.cfm">Login</a></li>
        <li><a href="register.cfm">Register</a></li>
      </ul>
    </div>
</div>
<div id="main">
  <div id="content">
  </div>
</div>
<div id="footer">&copy; #DateFormat(Now(), "yyyy")# Railo
Technologies</div>
```

```
</cfoutput>
</body>
</html>
```

2. Let's change the code in `layout.cfm` and add the following:

```
<cfif ThisTag.executionmode IS "start">
<!DOCTYPE html>
<html lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
  <link rel="stylesheet" href="css/main.css" type="text/css">
  <title>VideoShare</title>
</head>
<body>
<cfoutput>
<div id="header">
    <h1><a href="index.cfm">VideoShare</a></h1>
    <div id="topnav">
      <ul>
        <li><a href="login.cfm">Login</a></li>
        <li><a href="register.cfm">Register</a></li>
      </ul>
    </div>
</div>
</cfoutput>
<div id="main">
</cfif>

<cfif ThisTag.executionmode IS "end">
</div>
<cfoutput>
<div id="footer">&copy; #DateFormat(Now(), "yyyy")# Railo
Technologies</div>
</cfoutput>
</body>
</html>
</cfif>
```

3. As you can see, we have removed the `<div id="content"></div>` from the middle of the template and surrounded the top part (which every page will have) with the `<cfif thisTag.executionmode IS "start">` condition. This tells the Railo Server that it will only display that portion when the custom tag is opened and the bottom section will be called when the custom tag is closed. We also need to re-jig where we have our `<cfoutput>` tags so that they are within their respective `<cfif>` statements.

4. Let's change our `index.cfm` file to use this custom tag. Let's replace the contents of `index.cfm` with the following:

   ```
   <cf_layout>
     <div id="content">
     </div>
   </cf_layout>
   ```

What just happened?

Repeating code isn't very good, because you have to be tied down to doing multiple searches and replace actions when you want to change even the slightest thing. Custom tags are very handy for repeating code, as they are very light in performance and have other benefits that we will see later in this chapter, such as passing variables.

Registering users

All the actions that you can do in our VideoShare application will require users to be logged in, but before they can even log in, we should let users actually register. Before we create our registration form, we are going define the persistent objects of our application. Let's start with the user.

Time for action – creating our user model object

The `User` object in our application will need a few properties for us to work with, for example, a username, user e-mail address, and password.

1. Let's create a folder named `model` in our `videoshare` directory; this is where all our model objects are going to be stored so that they are out of the way, and we can easily tell where they are.

2. In the `model` folder, create a file named `User.cfc` and add the following content:

   ```
   component persistent="true"{
     property name="id" fieldtype="id" ormtype="int"
   generator="increment";
     property name="email";
     property name="password";
     property name="username";
   }
   ```

3. In the previous code, we see that we have used the `persistent="true"` property; this alerts the ORM in Railo Server that it should create a table for this component. Then, we use the `property name="id" fieldtype="id" ormtype="int" generator="increment";` to define a unique ID for this property of the name `id`. We then add properties for e-mail, password, and username. They will, by default, become `varchar` fields in our database.

4. Now that we have the object, let's create a form for people to register from. In the `videoshare` directory, create a template named `register.cfm` with the following code:

   ```
   <cf_layout>
   <cfparam name="FORM.username" default="">
   <cfparam name="FORM.email" default="">


   <h1>Register</h1>

   <cfoutput>
   <form action="register_user.cfm" method="post" accept-
   charset="utf-8">
     <div class="input">
       <label for="username">Username</label><input type="text"
   name="username" value="#form.username#" id="username">
     </div>
     <div class="input">
       <label for="email">Email</label><input type="email"
   name="email" value="#form.email#" id="email">
     </div>
     <div class="input">
       <label for="password">Password</label><input type="password"
   name="password" value="" id="password">
     </div>
     <div class="input">
   ```

```
  <label for="password_confirmation">Password Confirmation</
label><input type="password" name="password_confirmation" value=""
id="password_confirmation">
  </div>

  <p><input type="submit" value="Register &rarr;"></p>
</form>

</cfoutput>
</cf_layout>
```

5. There seems to be a lot going on here, but it's rather basic. We have used the `<cfparam>` tags to set default values for most of our fields (we ignored the password fields because we don't want to set a default for those) and we created a form. The form points to the `register_user.cfm` template that we are going to create next.

   We have also created a number of fields, and they have a default value by adding the `#form.username#` and `#form.email#`, so that if there is an error when we submit it, the user doesn't need to re-fill in the field.

6. You might have also noticed that we are now using our `<cf_layout>` tag surrounding the whole template. This saves us the trouble of re-writing the header again. Neat!

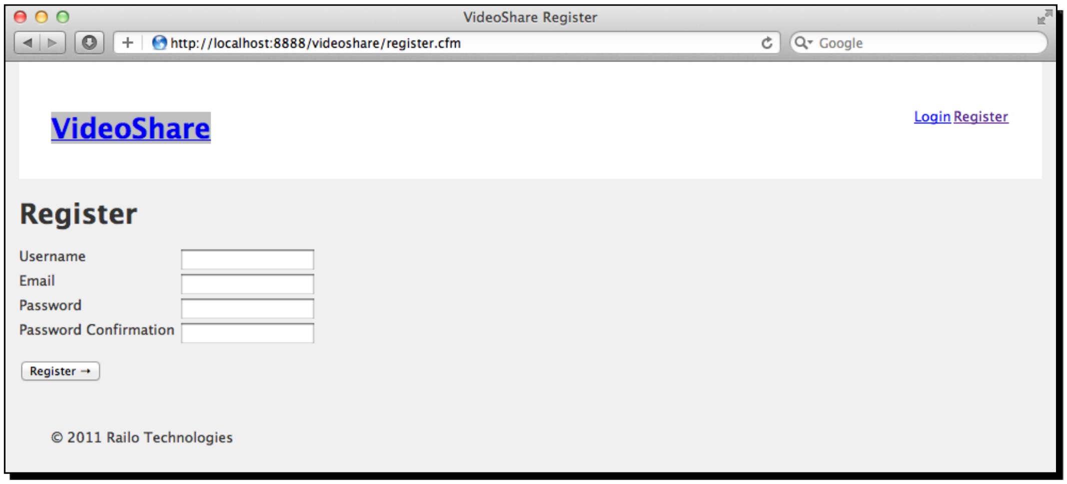
7. Before we go on, you might notice that the title of the page just says **VideoShare**. Let's improve our layout by adding the sub section that we are in. In the `register.cfm` file, change the code at the top to the following:

```
<cf_layout section="Register">
<cfparam name="FORM.username" default="">
<cfparam name="FORM.email" default="">
<h1>Register</h1>
```

8. We are now passing to our `layout.cfm` file an attribute called `section`. Let's edit the top of the `layout.cfm` template to display it:

```
<cfif ThisTag.executionmode IS "start">
<cfparam name="attributes.section" default="">
<!DOCTYPE html>
<html lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
  <link rel="stylesheet" href="css/main.css" type="text/css">
<cfoutput>  <title>VideoShare #attributes.section#</title> </
cfoutput>
</head>
```

9. As you can see, we used a `<cfparam>` to define the `attributes.section` variable; this variable is blank by default and then we add it to the `<title>` tag of our page, surrounding it in `<cfoutput>` tags so that it will be rendered. When we go to `http://localhost:8888/videoshare/register.cfm`, we now see the form as well as the title of the page, showing us that we are in the **Register** section of our site.



10. Now that we have the form ready, let's create the action template that will do all the work. Because our form was pointing to `register_user.cfm` for the action, let's create that template in our `videoshare` folder and put the following code in the beginning:

```
<cfparam name="FORM.username" default="">
<cfparam name="FORM.email" default="">
<cfparam name="FORM.password" default="">
<cfparam name="FORM.password_confirmation" default="">
<cfparam name="errors" default="#[]#">
<cfscript>

</cfscript>
<cf_layout section="Registered!">
<h1 id="congratulations!">Congratulations!</h1>
<p>You have successfully registered to Video Share!</p>
</cf_layout>
```

11. Not much here yet, as we are not actually processing, but you can see we are using `<cfparam>` to set up some defaults, including an array of errors, which we can pre-fill if anything went wrong. We also have a `<cfscript>` block. We are going to code our registration logic in `<cfscript>` here because it will be quicker to write without using tags.

12. Let's first check for some obvious errors in the registration page, such as making sure they filled in their names, that their e-mails are valid, and that their passwords match. In the `<cfscript>` block, add the following code:

```
<cfscript>
  if(!Len(FORM.username)){
    ArrayAppend(errors, "Username is missing");
  }
  if(!Len(FORM.email) OR !isValid("email", FORM.email)){
    ArrayAppend(errors, "Email is not valid");
  };

  if((!Len(FORM.password) OR !Len(FORM.password_confirmation)) or
FORM.password NEQ FORM.password_confirmation){
    ArrayAppend(errors, "The password was not set or it doesn't
match the confirmation password");
  }
</cfscript>
```

13. Next, we should check that they are not already registered. This is where we can see if there already is a user with this e-mail; it's quite simple, so let's add some code after the last check:

```
<cfscript>
  ...
  if((!Len(FORM.password) OR !Len(FORM.password_confirmation)) or
FORM.password NEQ FORM.password_confirmation){
    ArrayAppend(errors, "The password was not set or it doesn't
match the confirmation password");
  }

  //Check that the email is not already in the database;
  User = EntityLoad("User", {email=FORM.email});
  if(ArrayLen(User)){
    ArrayAppend(errors, "A user with this email address has
already registered! Try another email address.");

  }
</cfscript>
```

14. The User object is loaded by using the ORM function EntityLoad(); we pass in the object type we are expecting, User, and then pass in an array of conditions; in this case, email=FORM.email. If it returns any items in from the EnityLoad() function, it means that there are already existing users in the database, so we add another error to our error array.

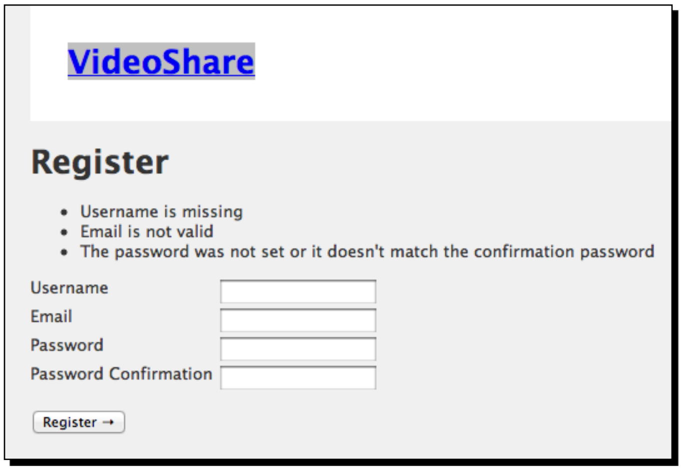
15. So what happens when you have an error? Simple, we include the register.cfm template again, by adding the following code:

```
<cfscript>

User = EntityLoad("User", {email=FORM.email});
if(ArrayLen(User)){
  ArrayAppend(errors, "A user with this email has already
registered! Try another email address");
}
if(ArrayLen(errors)){
  include template="register.cfm";
  abort;
}
```

Now that we are re-displaying the main register.cfm template when we have an error, let's display those errors in the register.cfm template. At the top of register.cfm, let's add another <cfparam> variable to set up our default errors array and then loop though them if there are any:

```
<cf_layout section="Register">
<cfparam name="FORM.username" default="">
<cfparam name="FORM.email" default="">
<cfparam name="errors" default="#[]#">

<h1>Register</h1>
<cfoutput>
  <cfif ArrayLen(errors)>
    <div class="errors">
      <ul>
         <cfloop array="#errors#" index="e">
           <li>#e#</li>
          </cfloop>
       </ul>
    </div>
  </cfif>
```

16. Great! Now, when we submit an empty form, we should get something similar to the following screenshot displayed:



Register

- Username is missing
- Email is not valid
- The password was not set or it doesn't match the confirmation password

17. So, now that we have managed to deal with erroneous form submissions, how about actually registering our user? After we check there aren't any errors, we can add the code to create a new user, since we are using the `abort;` function to stop the rest of the page processing and we are safe to add a new user. Let's add some code to `register_user.cfm` to handle that:

```
<cfscript>
 ...
  if(ArrayLen(errors)){
    include template="register.cfm";
    abort;
  }

  NewUser = EntityNew("user");
  NewUser.setEmail(FORM.email);
  NewUser.setPassword(FORM.password);
  NewUser.setUsername(FORM.username);
  EntitySave(NewUser);

  //Also log them in too!
  session.userid = NewUser.getID();
</cfscript>
```

18. To add a new user, we use the `EntityNew("User")` function to return a blank new ORM object for us to use. We then set the **Email**, **Password**, and **Username**. Finally, we use the `EntitySave(NewUser)` to persist our user to the database. In the final part of our function, we save the new user ID to the `SESSION` scope; this will tell our application that the user is logged in.

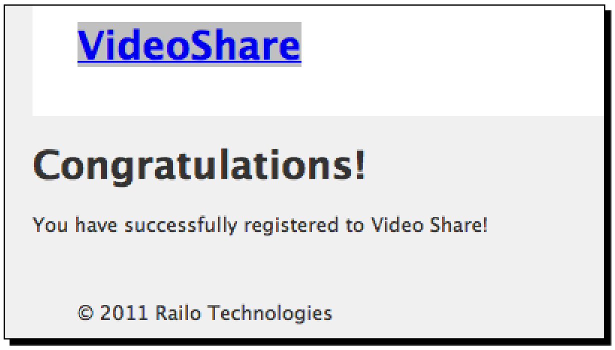
> The `EntityNew()` function takes an entity name as an input (in this case, `User`) and creates a new ORM object. The `EntitySave()` function will insert the `NewUser` object into the database. The beauty of the `EntitySave()` function is that Railo Server smartly figures out whether the record needs to be inserted or updated in the database. We can also tell the Railo Server if we want the current operation to be an insertion of a new record by using the 'true' parameter.

19. Finally, to tidy up our application, we should set a default variable in the `SESSION` scope, so that `session.userid` is always available. To do this, we can add it to the `onSessionStart()` function in our `Application.cfc` template. Let's add the following function so that our `Application.cfc` looks as follows:

```
component {
  this.name = "VideoShare";
  this.datasource = "VideoShare";
  this.ormEnabled = true;
  this.sessionmanagement = true;
  this.ormSettings = {};
  this.ormSettings.dbcreate = "dropCreate";

  function onSessionStart() {
    session.userid = 0;
   }
}
```

20. The code in `onSessionStart()` will make sure that every time we start a new session, the variable `session.userid` is available.

21. We can now go and add some real details to our form and submit it. We finally get the friendly welcome message that we had in our template:



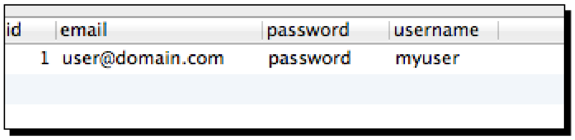
VideoShare

Congratulations!

You have successfully registered to Video Share!

© 2011 Railo Technologies

22. And by inspecting the database (you can use your favorite MySQL client or the command line application), we can see that the new user has been created.



id	email	password	username
1	user@domain.com	password	myuser

What just happened?

We managed to do a lot in this section. We created a form (with the default values for safety), added error handling by checking the user input from the registration form, checked our database for other users with the same e-mail address, created a new user using the `EntityNew()` function, and added the user ID to the session so that they are logged in after registering.

User login and logout

Now that the users are able to register, we should handle the login and logout, as well as check whether the users are indeed logged in or not. By now, we should be logged in (as we have just registered). Let's change the header to display whether we are logged in or not.

Time for action – log in or log out of the application

1. Open up the `layout.cfm` file and change the `header` section to look like the following code:

```
<body>
<cfoutput>
<div id="header">
    <h1><a href="index.cfm">VideoShare</a></h1>
    <div id="topnav">
      <ul>
      <cfif NOT session.userid>
        <li><a href="login.cfm">Login</a></li>
        <li><a href="register.cfm">Register</a></li>
      <cfelse>  <!--- only display if logged in --->
        <li><a href="logout.cfm">Logout</a></li>
        <li><a href="upload.cfm">Upload</a></li>
      </cfif>
      </ul>
    </div>
</div>
</cfoutput>
```

2. In the previously highlighted code, we are checking if `SESSION.userid` is set (to anything except `0`) and this means that a user has logged in. If they are logged in, we now display the **logout** and **upload** links; otherwise we display the **login** and **register** links.

3. When we now head to the home page of our VideoShare by going to `http://localhost:8888/videoshare/index.cfm`, we will see the **logout** and **upload** links. It's all working!

4. Let's add the logout functionality; all it has to do is set our `session.userid` variable to `0`. Create a `logout.cfm` template and put the following code to do this:

```
<cfset session.userid = 0>
<cf_layout section="Logout">
<h1>Sorry to see you go!</h1>
<p>You have successfully logged out of Video Share, come back
soon!</p>
</cf_layout>
```

5. The template is pretty simple. We are just setting `session.userid = 0` and displaying a friendly message to the user. That's it for logging the user out. Let's let them log in again.

6. First, we are going to create a `login.cfm` template with a login form asking our user for his/her e-mail and password. Add the following code into a new `login.cfm` template:

```
<cf_layout section="Login">
<cfparam name="FORM.email" default="">
<cfparam name="FORM.password" default="">
<cfparam name="errors" default="#[]#">
<cfoutput>
<h1 id="login">Login</h1>
<p>Enter your email and password to login!</p>


<cfif ArrayLen(errors)>
  <div class="errors">
    <ul>
      <cfloop array="#errors#" index="e">
        <li>#e#</li>
      </cfloop>
    </ul>
  </div>
</cfif>

<form action="login_user.cfm" method="post" accept-
charset="utf-8">
<div class="input">
<label for="email">Email</label>
<input type="email" name="email" value="#form.email#" id="email">
</div>
<div class="input">
<label for="password">Password</label>
<input type="password" name="password" value="" id="password">
</div>
<p><input type="submit" value="Login &rarr;"></p>
</form>

</cfoutput>
</cf_layout>
```

7. Even though there is a lot of code being added, you have seen most of it before with the registration form. We have our layout, our form variables which are set as default with the `<cfparam>` tag, a loop through any errors we encounter, and finally, the form with an e-mail and password field that we are sending to `login_user.cfm`.

8. Let's create the template that will actually log in our users. Let's create a `login_user.cfm` template and add the following code:

```
<cfparam name="FORM.email" default="">
<cfparam name="FORM.password" default="">
<cfparam name="errors" default="#[]#">

<cfscript>
  if(!Len(FORM.email) OR !isValid("email", FORM.email)){
    ArrayAppend(errors, "Email is not valid");
  };

  if(!Len(FORM.password)){
    ArrayAppend(errors, "The password was not set");
  }
  //Try to lookup the user
  User = EntityLoad("User", {email=FORM.email, password=FORM.
password}, true);

  if(isDefined("User")){
    session.userid = User.getID();
  }
  else {
    ArrayAppend(errors, "We couldn't find you, check your email
and password");
  }

  if(ArrayLen(errors)){
    include template="login.cfm";
    abort;
  }

</cfscript>


<cf_layout section="Login Welcome!">
<h1 id="welcome!">Welcome!</h1>
<p>You have logged in! You can upload videos now!</p>
</cf_layout>
```

9. Similar to our `register_user.cfm` template, we are first setting our default expected fields, checking for errors as we did before, and then using the `EntityLoad("User", {email=FORM.email, password=FORM.password}, true);` function, while passing in a structure of e-mail and password as our search criteria. The final `true` variable that we are passing into the `EntityLoad()` function is actually saying that the function should return only one record, rather than an array of objects. We then check if we have found a user whose e-mail and password matches what we have in our database, and if one is found, we set the `session.userid` variable to that user ID and display the welcome screen; otherwise, we add an error and include the login form again. Pretty simple right?

What just happened?

Using the same methods we saw in registering our users, we were able to set and unset the `SESSION.userid` variable that we are using in our application to say whether a user is logged in or not.

> **Other security functionality in Railo server**
>
> There are other tags that you can also use to log in and authenticate users in Railo Server. The tags `<cflogin>`, `<cfloginuser>`, and `<cflogout>` could accomplish the same thing. You can then use a number of functions to check if a user is logged in, such as `isUserLoggedIn()` and `isUserInRole()`.

Uploading videos

Now that we have our users nicely logged in, we can get to the core functionality of our application, namely, that of uploading and converting videos.

As was mentioned in *Chapter 7*, *Multimedia and AJAX*, we will be using the video extension you should have downloaded from the Extension store to convert uploaded videos and the `<cfvideoplayer>` tag to display it. We are also going to create some preview images for our videos, so when we display them, there will be a screenshot from the actual video to view.

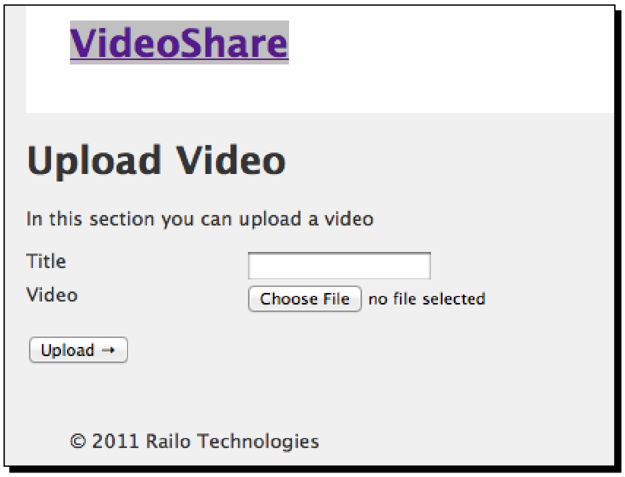
Time for action – uploading a video

1. Now that we are logged in and ready to upload a video, let's create a form for the user to do this. Add a template named `upload.cfm` in your `videoshare` folder and add the following code:

```
<cf_layout section="Upload Video">
<cfparam name="FORM.title" default="">
<cfoutput>
<h1 id="upload_video">Upload Video</h1>
<p>In this section you can upload a video</p>

<form action="upload_video.cfm" method="post" enctype="multipart/
form-data">
<div class="input">
<label for="title">Title</label>
<input type="text" name="title" value="#form.title#" id="title">
</div>
<div class="input">
<label for="video">Video</label>
<input type="file" name="video" id="video">
</div>


  <p><input type="submit" value="Upload &rarr;"></p>
</form>
</cfoutput>
</cf_layout>
```

2. This is a simple form. We have seen this before with the registration and login forms, but there are a couple of notable differences. The first difference being the `enctype="multipart/form-data"` attribute in the `<form>` tag. This is required to upload files (such as videos and pictures); otherwise, all that will be uploaded is the path to the video, not the video itself. The second difference is in regards to actually adding an `<input type="file" ...>` tag to upload our video.

3. When we head to `http://localhost:8888/videoshare/upload.cfm`, we can now see our neat little upload form.



VideoShare

Upload Video

In this section you can upload a video

Title

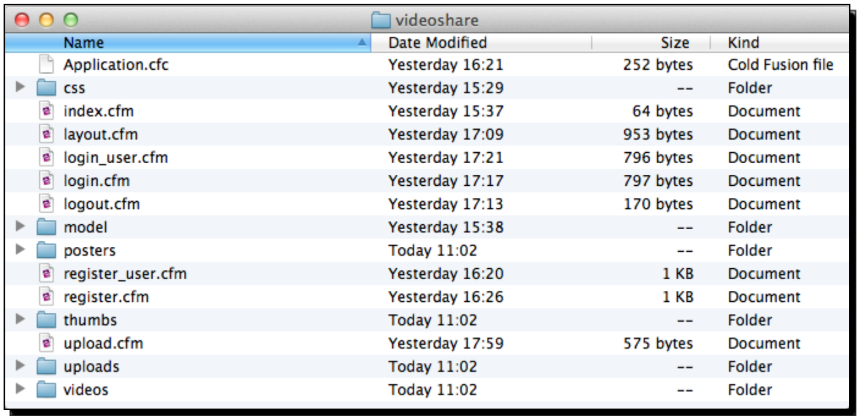
Video

© 2011 Railo Technologies

4. Now that we have our form, we need to create the file that is going to do the uploading for us; but before we do that, let's create four folders under our `VideoShare` directory:

   - `uploads`: Here, we are going to initially upload our videos
   - `videos`: Here, once converted, all the videos will reside
   - `thumbs`: Here we are going to place the thumbnails for our application
   - `posters`: Here we are going to put the large image of the videos that we are going to use as a poster frame

5. Your `VideoShare` folder should now look something like this:



videoshare

Name	Date Modified	Size	Kind
Application.cfc	Yesterday 16:21	252 bytes	Cold Fusion file
css	Yesterday 15:29	--	Folder
index.cfm	Yesterday 15:37	64 bytes	Document
layout.cfm	Yesterday 17:09	953 bytes	Document
login_user.cfm	Yesterday 17:21	796 bytes	Document
login.cfm	Yesterday 17:17	797 bytes	Document
logout.cfm	Yesterday 17:13	170 bytes	Document
model	Yesterday 15:38	--	Folder
posters	Today 11:02	--	Folder
register_user.cfm	Yesterday 16:20	1 KB	Document
register.cfm	Yesterday 16:26	1 KB	Document
thumbs	Today 11:02	--	Folder
upload.cfm	Yesterday 17:59	575 bytes	Document
uploads	Today 11:02	--	Folder
videos	Today 11:02	--	Folder

6. Let's upload the initial video. Under your `videoshare` folder, create a template named `upload_video.cfm`, add the layout, and so on, so that it looks like the following lines of code:

```
<cfparam name="FORM.title" default="">
<cfparam name="FORM.video" default="">
<cfparam name="errors" default="#[]#">


<cfscript>
  if(!Len(form.title) OR !Len(FORM.video)){
    ArrayAppend(errors, "Please Upload a valid video and give it a
title");
  include template="upload.cfm";
  abort;
  }
  file action="upload" destination="#expandPath("upl
oads")#" filefield="video" nameconflict="makeunique"
result="uploadedVideo";

</cfscript>

<cf_layout section="Video Uploaded!">
<h1 id="video_uploaded">Video Uploaded!</h1>
  <p>You have successfully uploaded your video! Our video
converting cats will get right on it to make your video look
awesome! </p>
</cf_layout>
```

7. In the previous code, we have our usual `<cfparam>` tags, making sure that we have defaulted the values of the `FORM` scope we are expecting, added an `error` array variable to store any errors, and then we checked that we have a title and video. The next part is important: `file action="upload" destination="#expandPath("uploads")#" filefield="video" nameconflict="makeunique" result="uploadedVideo";`; this line takes a file, and uploads it, and then we define the location as to where it will be uploaded with `destination="#expandPath("uploads")#"`. To tell Railo Server which form field we expect the file to come from, we add the `filefield="video"` attribute.

8. Because a lot of videos could be called the same, such as `MyVideo.mov`, we use the attribute `nameconflict="makeunique"` to make sure we have a unique name for this file when we save it. Finally, we return all the information using the `result="uploadedVideo"` attribute.

What just happened?

In this section, we set up a form as we had done before. However, this time we changed the `<form>` tag so that we could upload files to the Railo Server. When we post the form, we can then easily upload the file to a directory of our choice using the file tag.

Tags in CFScript

Railo Server is able to parse tags inside the `<cfscript>` tag as if they are scripts. We have just done that in the previous section. For example, we can write the following tag:

```
<cffile action="upload" destination="#expandPath("upl
oads")#" filefield="video" nameconflict="makeunique"
result="uploadedVideo">
```

This is done inside the `<cfscript>` block as:

```
file action="upload" destination="#expandPath("uplo
ads")#" filefield="video" nameconflict="makeunique"
result="uploadedVideo";
```

We do this by simply removing the first `<cf` part of the string and replacing the closing angle bracket (`>`) with a semi-colon (`;`). You can convert nearly all of the Railo Server tags to the `<cfscript>` format.

Adding security

One of the things we haven't touched yet is one of the security requirements we have for users to upload a file. Only users that have registered and logged in should be able to upload a file, and presently, this is not the case. Let's add another custom tag to protect these templates.

Time for action – adding the secure tag

To add some security, we need to check if a user has logged in; we can do this by using a tag that can be added at the top of each file that we want to secure. Let's add a custom tag called `<cf_secure>` to our templates:

1. Create a template under the `videoshare` directory named `secure.cfm` and add the following code:

```
<cfif session.userid NEQ 0>
  <cfinclude template="login.cfm">
  <cfabort>
</cfif>
```

2. The tag is rather simple; it checks whether we have a `session.userid` set and makes sure it isn't equal to zero with the `NEQ` operator. If it is not set, it includes our `login.cfm` template and then aborts the rest of the rendering of the page. If there is a `session.userid` set, it will ignore this statement.

3. At the top of the `upload.cfm` file, let's call the `<cf_secure>` tag:

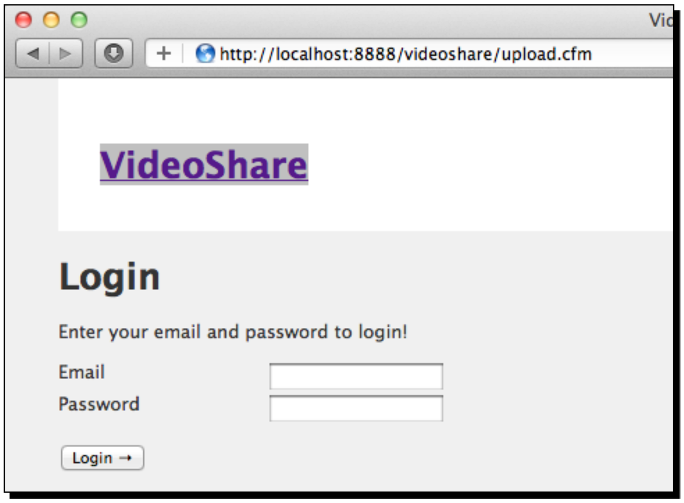
```
<cf_secure>
<cf_layout section="Upload Video">
<cfparam name="FORM.title" default="">
<cfoutput>
<h1 id="upload_video">Upload Video</h1>
```

4. When a user of our site now requests `http://localhost:8888/videoshare/upload.cfm` and they haven't logged in, they will now get the `login.cfm` template instead of the one they requested.

5. Let's do the same for the `upload_video.cfm` template:

```
<cf_secure>
<cfparam name="FORM.title" default="">
<cfparam name="FORM.video" default="">
<cfparam name="errors" default="#[]#">
<cfscript>
...
```

6. If we now try going to `http://localhost:8888/videoshare/upload.cfm`, we will get a login screen:



Login

Enter your email and password to login!

What just happened?

The requirements of our application are such that there are pages that need to be protected. By creating a simple tag that does the protection for us, we can change how we handle security at a later stage, update the `<cf_secure>` tag, and know that it will all still work as required.

We could have added the code to check if a user is logged in, in each of our templates, but it is always better to externalize code that is going to be repeated.

Assigning videos to users

In our application, we will want to know who has uploaded which video. We are also going to need to store a reference to our video in the database. What should our Video model object look like?

If we think about it for a moment, we can come up with a list of properties for a video that will be displayed:

- **Title**: The name of the video that the user has given us
- **Original File**: The name of the original file that was uploaded
- **File**: A unique name for the converted video, its thumbnail, and poster image
- **Status**: Whether the video has been converted or not
- **Converted At**: The time when the video was converted
- **Uploaded At**: The time when the video was uploaded
- **User**: The user that uploaded the video

With this information, let's get on and create our persistent object for the video.

Time for action – storing our video to the database

1. Let's create a `Video.cfc` file in our `model` folder under the `videoshare` folder and add the following code to the template:

```
component persistent="true"{
  property name="id" fieldtype="id" ormtype="int"
generator="increment";
  property name="title";
  property name="status" ormtype="int" default="0";
  property name="originalfile";
  property name="file";
  property name="convertedAt" ormtype="timestamp";
  property name="uploadedAt" ormtype="timestamp";
  property name="User" fieldtype="many-to-one" cfc="User"
fkColumn="User_Id";
  }
```

2. Working through the code, we can see it's very similar to the User object. It has an id property (the primary key), but we can see we have other types of properties that are defined by the ormtype attribute. For example, our status property is defined as an integer with a default of 0 with ormtype="int" default="0" and our convertedAt and uploadedAt properties are defined as date/time fields using the ormtype="timestamp" attribute.

   Finally, we have a property that is defined as a User. Here we define that many videos can be uploaded by one user using the fieldtype="many-to-one". This is achieved by adding a column in the Video table called User_Id, which represents the ID of the user.

3. Because we can relate many videos to one user, we should also update the User to say that they can have many videos. So let's add that property to the User.cfc object in our model directory:

```
component persistent="true"{
  property name="id" fieldtype="id" ormtype="int"
generator="increment";
  property name="email";
  property name="password";
  property name="username";
  property name="videos" fieldtype="one-to-many" cfc="Video"
fkColumn="User_Id";
}
```

4. You can see that we have now created a one-to-many relationship between a User object and a Video object by using the fieldtype="one-to-many" attribute.

5. Now that we have defined our relationships, we can upload the video and save it in the database. In the upload_video.cfm template, we can add the following code:

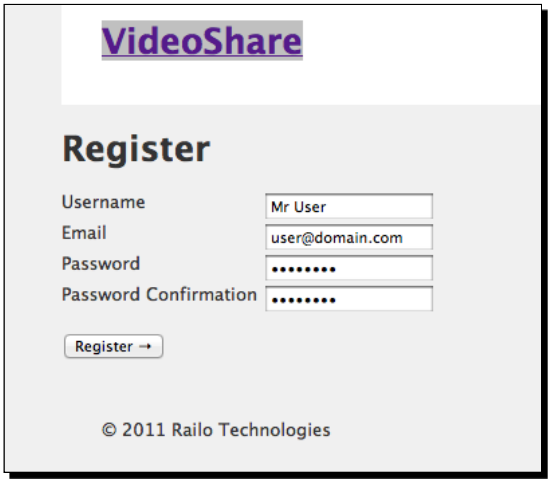
```
<cf_secure>
<cfparam name="FORM.title" default="">
<cfparam name="FORM.video" default="">
<cfparam name="errors" default="#[]#">
<cfscript>
  if(!Len(form.title) OR !Len(FORM.video)){
    ArrayAppend(errors, "Please Upload a valid video and give it a
title");
  include template="upload.cfm";
  abort;
  }
  file action="upload" destination="#expandPath("upl
oads")#" filefield="video" nameconflict="makeunique"
result="uploadedVideo";
```

```
    User = EntityLoad("User", session.userid, true);
    VideoObj = EntityNew("Video");
    VideoObj.setTitle(FORM.title);
    VideoObj.setOriginalFile(uploadedVideo.serverfile);
    VideoObj.setUser(User);
    VideoObj.setUploadedAt(Now());

    EntitySave(VideoObj);
  </cfscript>
```

6. Because we want to add a user to the new `Video` object we are storing, we first load the `User` object by using the `EntityLoad("User", session.userid, true)` function and by getting the `userid` from the `SESSION` scope. Then, we create a new `Video` object using the `EntityNew("Video")` function. Each property will have a `set<PropertyName>` method automatically created in our object, so we can set the `title`, the `originalFile`, the `UploadedAt` time (using the `Now()` function), and finally we set the `User` object we loaded previously.

   To save the object, we simply call the `EntitySave(VideoObj)` method and our object will be stored in the database.

7. We can now try to upload a video. Because we are using the `DropCreate` property in the `this.ormsettings.dbcreate` property in `Application.cfc`, we might have to re-register our user:



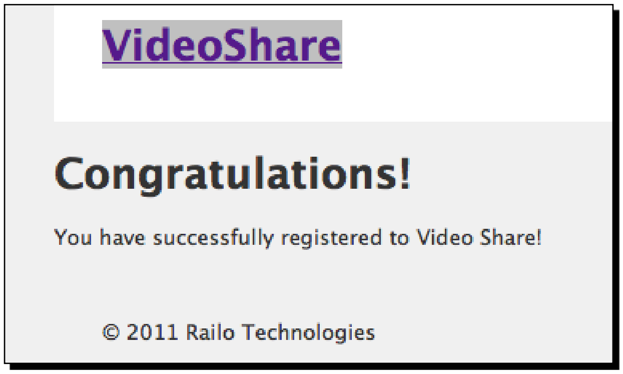
VideoShare

Register

Username: Mr User
Email: user@domain.com
Password: ••••••••
Password Confirmation: ••••••••

© 2011 Railo Technologies

8. Once we have done that, we should get a friendly greeting stating that we have registered, and we should also be logged in:



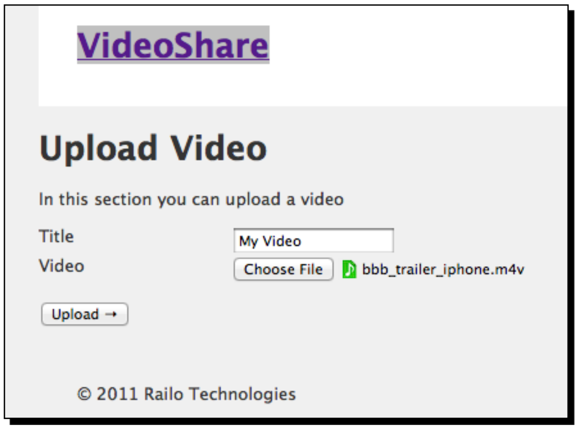
VideoShare

Congratulations!

You have successfully registered to Video Share!

© 2011 Railo Technologies

9. Now we can click **upload**, set the name of our video, and select an appropriate video to upload (we chose the trailer of Big Buck Bunny from `http://www.bigbuckbunny.org/`):



VideoShare

Upload Video

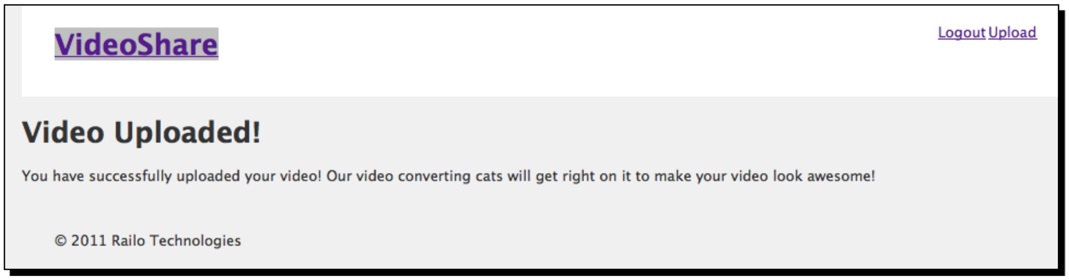
In this section you can upload a video

Title My Video

Video Choose File bbb_trailer_iphone.m4v

© 2011 Railo Technologies

10. Once we uploaded it, we got another friendly greeting stating that we have uploaded the file:



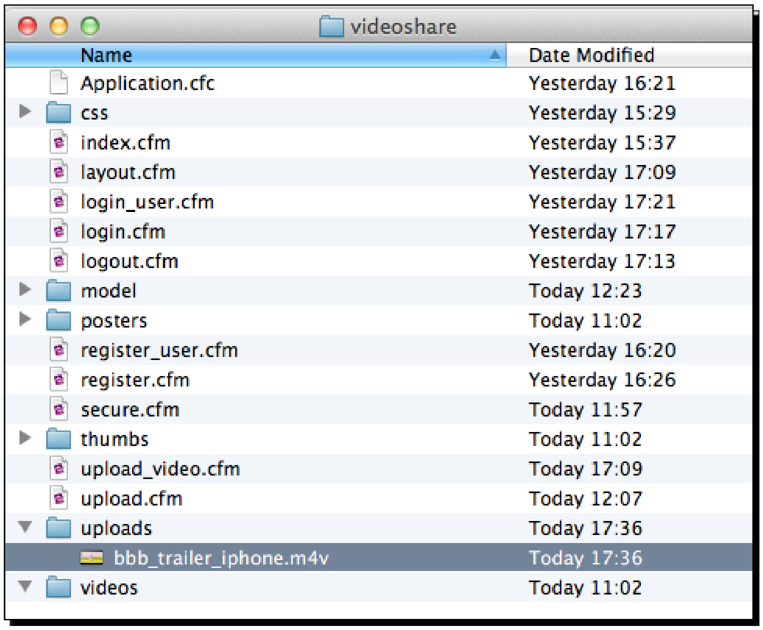
VideoShare

Video Uploaded!

You have successfully uploaded your video! Our video converting cats will get right on it to make your video look awesome!

© 2011 Railo Technologies

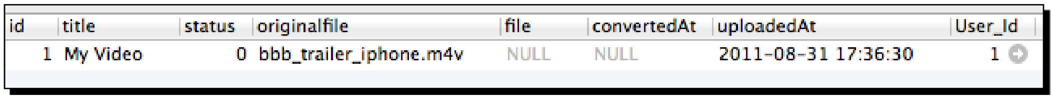
11. If we now look in the `uploads` folder of our `videoshare` directory, we can see that the file has been uploaded! Success.



videoshare

Name	Date Modified
Application.cfc	Yesterday 16:21
css	Yesterday 15:29
index.cfm	Yesterday 15:37
layout.cfm	Yesterday 17:09
login_user.cfm	Yesterday 17:21
login.cfm	Yesterday 17:17
logout.cfm	Yesterday 17:13
model	Today 12:23
posters	Today 11:02
register_user.cfm	Yesterday 16:20
register.cfm	Yesterday 16:26
secure.cfm	Today 11:57
thumbs	Today 11:02
upload_video.cfm	Today 17:09
upload.cfm	Today 12:07
uploads	Today 17:36
bbb_trailer_iphone.m4v	Today 17:36
videos	Today 11:02

12. If we now check our database, we can see that a `Video` table has been created, and also a new entry has been placed for our video that relates it back to our user via the `User_Id` column.



id	title	status	originalfile	file	convertedAt	uploadedAt	User_Id
1	My Video	0	bbb_trailer_iphone.m4v	NULL	NULL	2011-08-31 17:36:30	1

What just happened?

In this section, we used the ORM `fieldtype` to define a one-to-many relationship between the `User` and `Video` objects, as well as relating many `Videos` to one `User` using the `one-to-many` relationship. Finally, once we uploaded the video, we were able to simply store the video in the database and relate it to the user, without having to write a single line of SQL code.

DropCreate and other ormsettings.dbcreate properties

In the Application.cfc template, we set the this.ormsettings.dbcreate = "DropCreate" property. This property tells Railo Server that all the tables should be dropped (deleted) whenever there is a change to any of the model files. The downside of this is that all your data will also be deleted. This could be fine in development, but can get rather annoying. You have other choices though; you can set this property to:

None: It means that none of the tables will be altered, regardless of changes to your model objects.

Update: Using this, new tables will be created and any additional columns will be added, hence maintaining your data.

DropCreate: This will delete the tables and recreate them whenever there is a change or Railo Server is re-started.

Converting and playing videos

So far, we have uploaded our videos and added them to the database, but now, we should convert the video to a format that is useful for viewing using the <cfvideoplayer> tag, so that users of the VideoShare site can actually play the videos. After all, it that is the whole point of the application. Luckily, this is an easy task with the <cfvideo> tag. Let's convert the videos we uploaded to a format that is actually useful.

Time for action – converting the uploaded video

Let's edit the video_upload.cfm file to add the code required to convert this video to an Internet-viewable format.

1. In the video_upload.cfm file, in our VideoShare folder, add the following lines of code; we will go through what they do in a second:

```
<cfscript>
...
  VideoObj.setUser(User);
  VideoObj.setUploadedAt(Now());
  EntitySave(VideoObj);

  newName = CreateUUID();
  videoName = newName & ".flv";
</cfscript>
<cfvideo   action="convert"
```

```
              profile="internet"
              source="uploads/#VideoObj.getOriginalFile()#"
              destination="videos/#videoName#">
<cfscript>
  VideoObj.setFile(newName);
  VideoObj.setConvertedAt(Now());
  VideoObj.setStatus(1);
  EntitySave(VideoObj);
</cfscript>

<cf_layout section="Video Uploaded!">
```

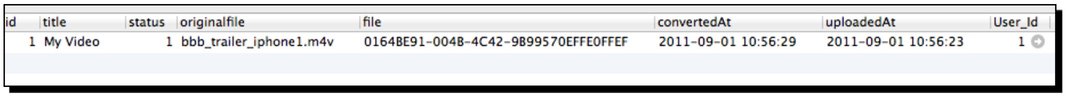
2. In the previous code, we started by creating the `newName` variable for our video. This is done by using the `CreateUUID()` function, which will give us a unique string for our converted video. This is a shortcut, but you could use any unique name. In this case, it will produce a name like **0164BE91-004B-4C42-9B99570EFFE0FFEF**.

   We then set the `videoName` variable to be the `newName`, but we add the **.flv** file extension.

   We break out of `<cfscript>` since the `<cfvideo>` tag is one of the tags that doesn't have a `<cfscript>` version equivalent (this will change in the future) and set the `action="convert"`. We then use the video format shortcut to say we want an Internet-formatted video with the `profile="internet"` attribute. Finally, we get the original filename from our saved `Video` object, set it as the source attribute in `source="uploads/#VideoObj.getOriginalFile()#"`, and set the destination with the new name we have created.

   We then add some more properties to our `Video` object: we set the file attribute to the new name (that we created with a `CreateUUID()` function), set the date we converted it, and set the `status` to 1. We are going to be using the status to get all the "published" videos later on.

3. If we look in our `videos` folder, where we store our converted files, you will now see a file named **0164BE91-004B-4C42-9B99570EFFE0FFEF.flv**. Also, if you look in the database, the entry for the video should be updated.

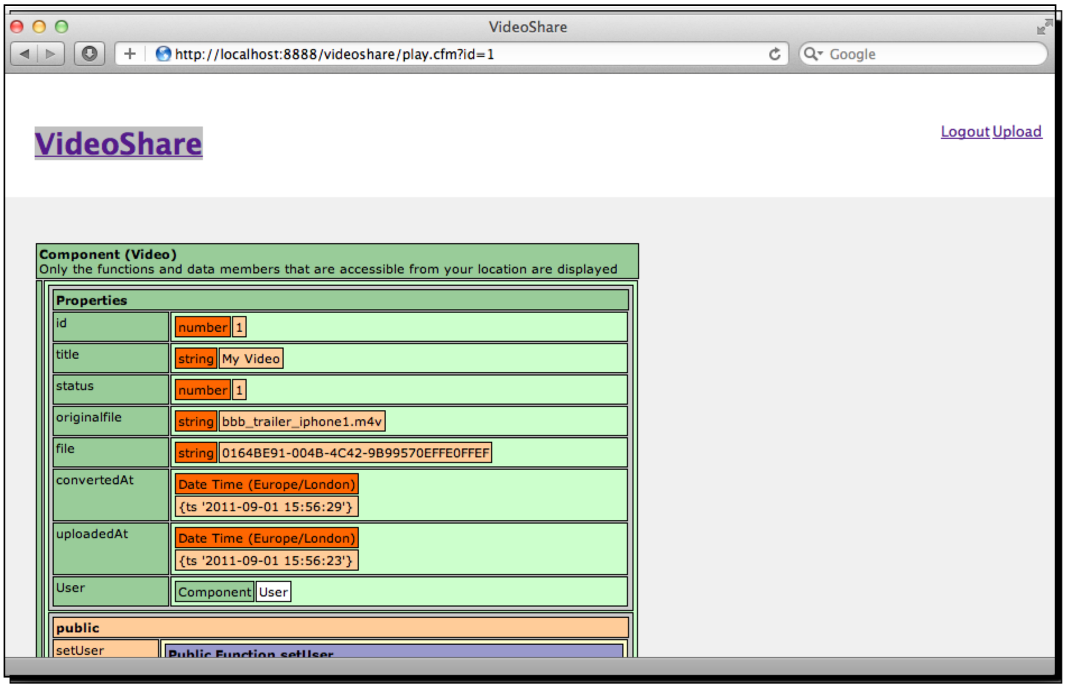


id	title	status	originalfile	file	convertedAt	uploadedAt	User_Id
1	My Video	1	bbb_trailer_iphone1.m4v	0164BE91-004B-4C42-9B99570EFFE0FFEF	2011-09-01 10:56:29	2011-09-01 10:56:23	1

4. So far, we have managed to convert the video, but now we should create a page to display it. Let's create a new page named `play.cfm`, which will be used to play our videos and put the following code:

```
<cf_layout>
  <cfparam name="URL.id" default="0">
  <div id="content">
    <cfset Video = EntityLoad("Video", {status=1, id=URL.id},
true)>
    <cfdump var="#Video#">

  </div>
</cf_layout>
```

5. This file has a `<cfparam>` at the top, so that we can define a default `URL` variable of `id`. We then use the `EntityLoad("Video", {status=1, id=URL.id}, true)` function to load a video with a status of `1` and an ID which is the `URL.id` that we passed in. This should return a `Video` object if found. Finally, we added `<cfdump var="#Video#">`, so that we can see the data object that is passed back:

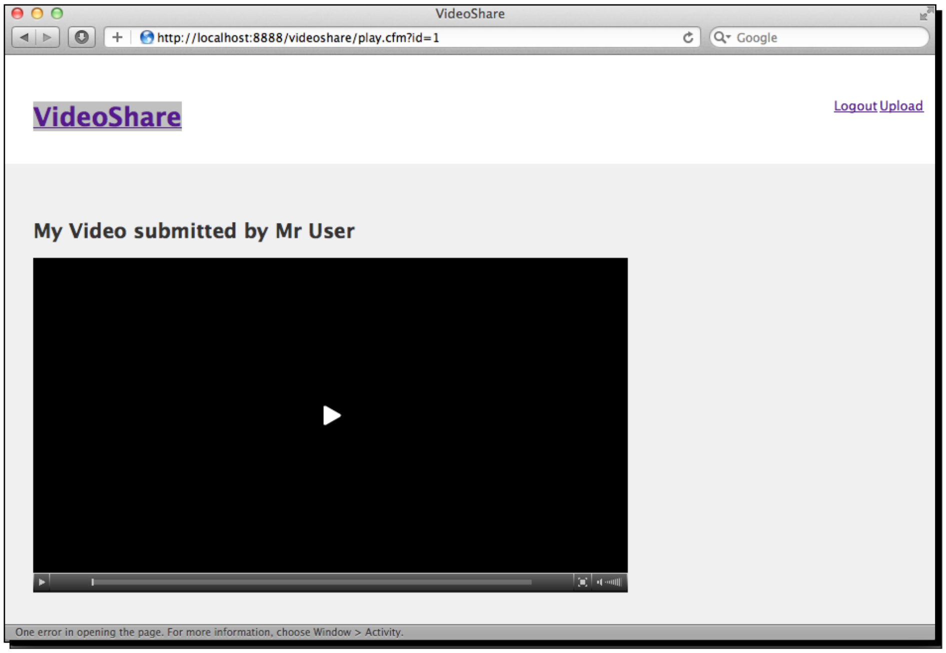


```
Component (Video)
Only the functions and data members that are accessible from your location are displayed
Properties
id            number 1
title         string My Video
status        number 1
originalfile  string bbb_trailer_iphone1.m4v
file          string 0164BE91-004B-4C42-9B99570EFFE0FFEF
convertedAt   Date Time (Europe/London)
              {ts '2011-09-01 15:56:29'}
uploadedAt    Date Time (Europe/London)
              {ts '2011-09-01 15:56:23'}
User          Component User
public
setUser       Public Function setUser
```

6. Great! We got the video we uploaded back. Now, let's remove the `<cfdump>` tag and replace it with some code to display the video:

```
<cf_layout>
  <cfparam name="url.id" default="0">
<cfoutput>
  <div id="content">
    <cfset Video = EntityLoad("Video", {status=1, id=url.id},
true)>
    <h2>#Video.getTitle()# submitted by #Video.getUser().
getUserName()#</h2>
    <cfvideoplayer video="videos/#Video.getFile()#.flv"
width="600" height="338">
  </div>
</cfoutput>
</cf_layout>
```

7. First, we surround our code with a `<cfoutput>` tag; this will allow us to output the variables. Then, we set the header, while getting the title of the video, and as we have a `User` object in our `Video` object, we can get the username of the person that posted it by using the syntax `Video.getUser().getUserName()`. Finally, we use the `<cfvideoplayer>` tag to display the video, setting the `source` to the `videos` directory and adding the filename and extension (as well as the width and height of the player), we can now view the video by going to `http://localhost:8888/videoshare/play.cfm?id=1`:



My Video submitted by Mr User

One error in opening the page. For more information, choose Window > Activity.

8. When we press play, we get our video playing.

What just happened?

This section covered a lot of ground, but we have seen most of the functions before. We started out by using the `<cfvideo action="convert">` tag to convert our video to a `.flv` file. We then used the ORM to update the `Video` object we created before with the new name and status.

Finally, we used the `EntityLoad()` function to load our video object from the database to get the information to pass to the `<cfvideoplayer>` tag and display our video.

Video format shortcuts

You might be wondering where the value for the `<cfvideo>` tag's format attribute comes from? When you installed the CFVideo Extension, Railo Server created a file that lists various formats; you can find this file in `<Railo Install Directory>/webroot/WEB-INF/railo/video/video.xml`.

This file lists a number of different video format profiles that you can use. You can, in fact, copy the attributes of each profile and put them into the `<cfvideo>` tag individually to refine your conversions, or add new ones that are specific to your needs.

Creating thumbnails for our videos

At the moment, before you click play in the `play.cfm` page, our video just displays a black screen. This isn't very friendly, so maybe we should add some images to display as a preview.

This is easy with the `<cfvideo>` tag as we have another action, called `cutImage`. Let's use this tag to create some thumbnails and a preview image.

Time for action – creating images from a video

Let's go add the ability to create thumbnails from our video.

1. Let's open up `upload_video.cfm` again, and add the following code after we convert the video:

```
  newName = CreateUUID();
  videoName = newName & ".flv";
  imageName = newName & ".jpg"
</cfscript>
  <cfvideo action="convert"
```

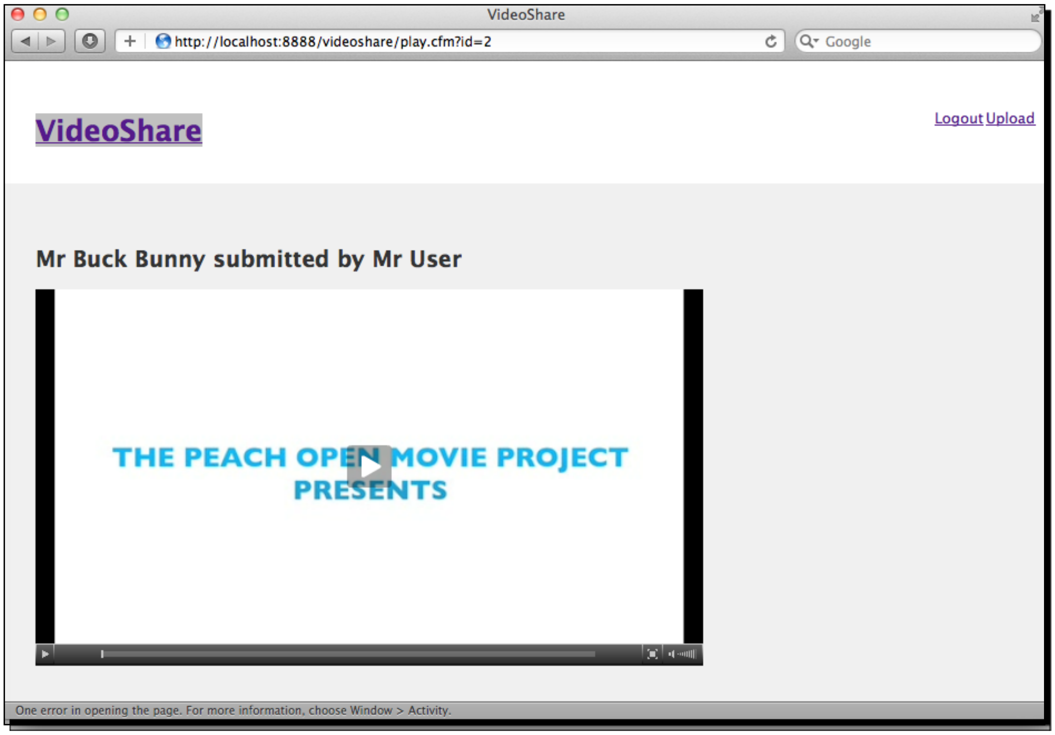
```
            profile="internet"
            source="uploads/#VideoObj.getOriginalFile()#"
            destination="videos/#videoName#">

        <cfvideo action="cutimage"
            source="uploads/#VideoObj.getOriginalFile()#"
             start="1s"
             destination="thumbs/#imageName#" width="100">
        <cfvideo action="cutimage"
            source="uploads/#VideoObj.getOriginalFile()#"
            start="1s"
             destination="posters/#imageName#">
```

2. In the previous code, we have added a new variable named `imageName`, which we have created using `newName`, and appended the `.jpg` extension. This is what we are going to call our images. Then, after we convert the video, we use the `<cfvideo>` tag again, but this time, we will do it by using `action="cutimage"` and passing in the original video as the source, and by setting where in the video we want to get an image from. In this case, we use `start="1s"` to get a frame from one second into the video. We then set the destination where we want to save this file. In the first instance, we use the thumbs directory, as we then use the `width="100"` attribute to make an image `100` pixels in width. The `<cfvideo>` tag is intelligent enough to figure out the height of the video. We then repeat this tag again, but we remove the width and save it in the `posters` folder. We now have a 100 pixel-wide thumbnail in the `thumbs` folder and a large image in the `posters` folder.

3. Now that we have created our poster image, let's edit the `play.cfm` template and add a placeholder to the `<cfvideoplayer>` tag:

```
<cf_layout>
  <cfparam name="url.id" default="0">
<cfoutput>
  <div id="content">
    <cfset Video = EntityLoad("Video", {status=1, id=url.id},
true)>
    <h2>#Video.getTitle()# submitted by #Video.getUser().
getUserName()#</h2>
<cfvideoplayer video="videos/#Video.getFile()#.flv"
        width="600" height="338"
     preview="posters/#Video.getFile()#.jpg">
  </div>
</cfoutput>
</cf_layout>
```

4. All we have to do to display a preview frame is add the `preview="posters/#Video.getFile()#.jpg"` attribute to the `<cfvideoplayer>` tag. Now, when we upload a new video, you should see a nice preview of the video as shown next:



VideoShare
http://localhost:8888/videoshare/play.cfm?id=2
Google
VideoShare
Logout Upload
Mr Buck Bunny submitted by Mr User
THE PEACH OPEN MOVIE PROJECT PRESENTS
One error in opening the page. For more information, choose Window > Activity.

What just happened?

The `<cfvideo>` tag has a number of actions, as we saw in *Chapter 7, Multimedia and AJAX*; one of them being the ability to get an image still from any part of a video. In this section, we cut out an image from the video at one second into the clip (most movies start with a blank screen, so this was a good compromise) and then we added it to the `<cfvideoplayer>` tag using the `preview` attribute.

Our application is certainly coming along! Let's add some user interaction now.

Adding comments to our video page

When users are viewing a video, it would be great if they could add a comment, but of course, only if they are logged in.

In this section, we are going to follow on what we have been doing so far and add the ability for a user to comment on an individual video. Let's get started by creating a comment-persisted component.

Time for action – adding comments to our videos

1. In our `model` folder, which is under the `videoshare` directory, let's create a new component named `Comment.cfc` and add the following code to define it:

```
component persistent="true" {
    property name="id" fieldtype="id" ormtype="int"
generator="increment";
    property name="comment";
    property name="postedAt" ormtype="timestamp";
    property name="User" fieldtype="many-to-one" cfc="User"
fkColumn="User_Id";
    property name="Video" fieldtype="many-to-one" cfc="Video"
fkColumn="Video_Id";
}
```

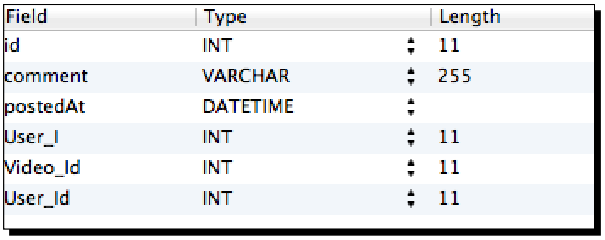
2. In the previous code, we have defined a comment as having an `id`, a `comment` property, and the date it was posted. We have also defined its relationships to the other model objects in our application. A user can have many comments; therefore, many comments can belong to one user. The same goes for a video; a video can have many comments, but a comment can only belong to one user. We define these relationships easily using the `fieldtype="many-to-one"`.

> **fieldtype and ORM**
>
> The `fieldtype` attribute can be used to define the relationship between two ORM objects (or database tables). In the previous code, we had two properties that were defined with the `fieldtype` attribute with a `many-to-one` relationship between the `User` and `Video` using the columns `User_Id` and `Video_Id`. The relationship types that are allowed are `one-to-one`, `one-to-many`, `many-to-one`, and `many-to-many`.

3. Because we have defined relationships from the comment to the user, let's define the relationship back to the comment from the user. Let's add the following to the `User.cfc` in our `model` directory:

```
component persistent="true"{
  property name="id" fieldtype="id" ormtype="int"
generator="increment";
  property name="email";
  property name="password";
  property name="username";
  property name="videos" fieldtype="one-to-many" cfc="Video"
fkColumn="User_Id";
  property name="comments" fieldtype="one-to-many" cfc="Comment"
fkColumn="User_Id";
}
As you can see we have added a new property for the comment with a
fieldtype="one-to-many"
Let's add the relationship to the Comments object in the Video.cfc
too:
component persistent="true"{
    property name="id" fieldtype="id" ormtype="int"
generator="increment";
    property name="title";
    property name="status" ormtype="int" default="0";
    property name="originalfile";
    property name="file";
    property name="convertedAt" ormtype="timestamp";
    property name="uploadedAt" ormtype="timestamp";
    property name="User" fieldtype="many-to-one" cfc="User"
fkColumn="User_Id";
    property name="Comments" fieldtype="one-to-many" cfc="Comment"
fkColumn="Video_Id";
  }
```

4. This is nearly identical to the User component, except we change the `fkColumn` to use `Video_Id` as the foreign key. After we try this code again, we can have a look at our database. We should have a Comment table with the `video_id` and `user_id` columns:



Field	Type	Length
id	INT	11
comment	VARCHAR	255
postedAt	DATETIME	
User_I	INT	11
Video_Id	INT	11
User_Id	INT	11

5. Now that we have defined our relationships, let's add the code to display comments in the `play.cfm` file:

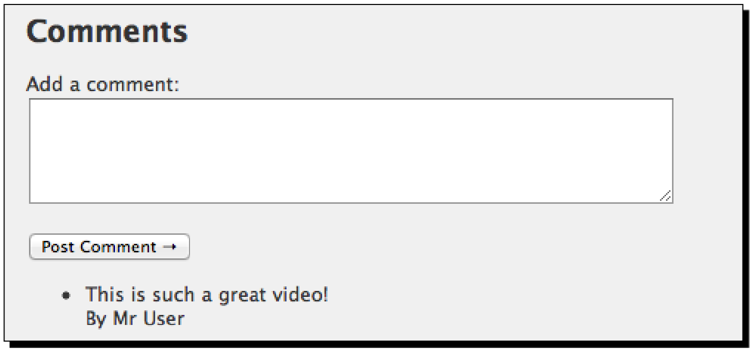
```
  <div id="content">
    <cfset Video = EntityLoad("Video", {status=1, id=url.id},
true)>
    <h2>#Video.getTitle()# submitted by #Video.getUser().
getUserName()#</h2>
    <cfvideoplayer video="videos/#Video.getFile()#.flv"
width="600" height="338"
              preview="posters/#Video.getFile()#.jpg">
  </div>

  <div id="comments">
    <h2>Comments</h2>
    <div id="commentform">
      <form action="comment_add.cfm" method="post" accept-
charset="utf-8">
        <input type="hidden" name="videoid" value="#Video.
getID()#">
        <label for="comment:">Add a comment:</label><br>
        <textarea name="comment" cols="60" rows="5"></textarea>
        <p><input type="submit" value="Post Comment &rarr;"></p>
      </form>
    </div>
    <cfset comments = Video.getComments()>
    <ul class="commentList">
      <cfloop array="#Comments#" index="comment">
        <li>#comment.getComment()#<br>By #comment.getUser().
getUserName()#</li>
      </cfloop>
    </ul>
  </div>
```

6. After we display the video, we add a new `<div>` for the comments and create a normal form that is posting to `comment_add.cfm` (we shall create this file in a minute). In that form, we have a hidden field that has the video ID defined as `<input type="hidden" name="videoid" value="#Video.getID()#">`. After the form, we set a variable called `comments` by calling the `Video.getComments()` method from the `Video` object. Railo Server had created this method when we defined the relationships. We then use the `<cfloop array="#Comments#" index="comment">` to loop through the comments array and display them in a list. Because each comment will have a related user, we get the username by calling the `comment.getUser().getUserName()` on each comment.

7. Let's handle the actual posting of a comment by creating the `comment_add.cfm` template with the following code:

```
<cf_secure>
<cfscript>
  param name="FORM.comment" default="";
  param name="FORM.videoid" default="0";

  Comment = EntityNew("Comment");
  Video = EntityLoad("Video", FORM.videoid, true);
  User = EntityLoad("User", SESSION.userid, true);
  Comment.setUser(User);
  Comment.setVideo(Video);
  Comment.setComment(FORM.comment);
  Comment.setPostedAt(Now());
  EntitySave(Comment);

  location url="play.cfm?id=#FORM.videoid#" addtoken=false;
</cfscript>
```

8. In this template, we have added the `<cf_secure>` tag at the top, since you need to be logged in to actually post. Then we have set defaults for the `FORM.comment` and `FORM.videoid` variables using the `<cfscript>` version of the `<cfparam>` tag. After that, we create a new `Comment` object using the `EntityNew("Comment")` function, load up our `Video` object and `User` object, set them to the `Comment` object, set the comment that a user has posted and when it was posted, and then save the comment using the `EntitySave(Comment)` function. Finally, we use the location `url="play.cfm?id=#FORM.videoid#" addtoken=false;` to redirect us back to the video. If we now go and add a comment on our video, we should see something like:



Comments

Add a comment:

- This is such a great video!
  By Mr User

What just happened?

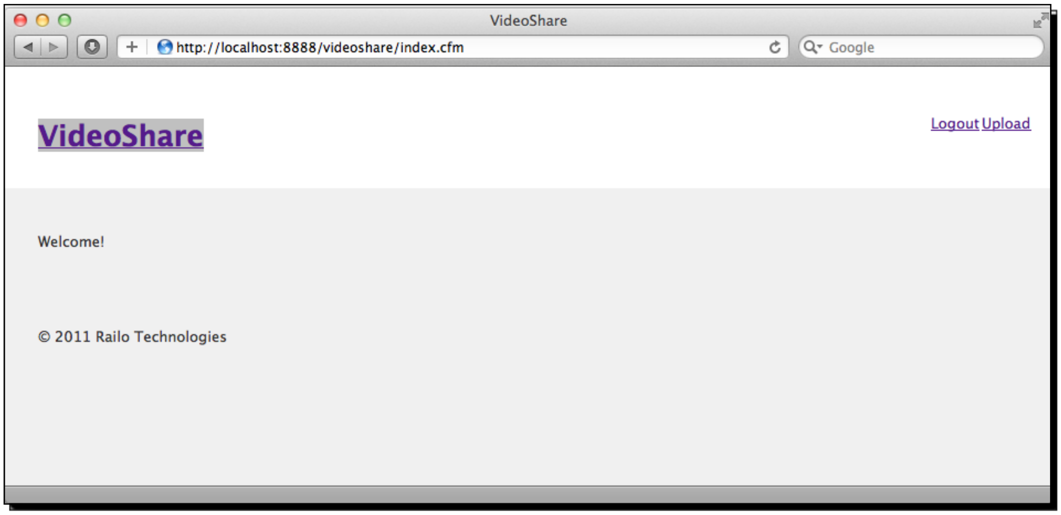
Continuing on from what we have learned about ORM relationships, we created a Comment object that was related to the `User` and `Video` objects. We then modified the User and Video objects to also have relationships with comments. This is useful in our video display page as we want to be able to display any comment related to that video. Finally, we added a page that allowed the saving of a comment by adding other objects to it.

Creating the home page

So far we have created a page to display a single video by its ID. But how do we find our videos in the first place? You might also ask, why did we create thumbnails for our videos if we are not using them?

In this section, we are going to change our home page to display a video and the most recent videos that have been uploaded as well as add a list of related videos that will use the thumbnails.

Let's remind ourselves what our home page currently looks like by going to `http://localhost:8888/videoshare/index.cfm`:



VideoShare

Logout Upload

Welcome!

© 2011 Railo Technologies

It looks a bit boring, doesn't it? We should improve it by displaying the latest video which is ready to play, along with a list of thumbnails of all the latest videos. That should make it a lot more interesting.

Time for action – getting the latest videos

We could go and get the latest video, then do another query to get a list of the latest videos. Instead of doing that, we could simply get the top ten latest videos and use the first one as our "hero" video in the middle. Let's do that:

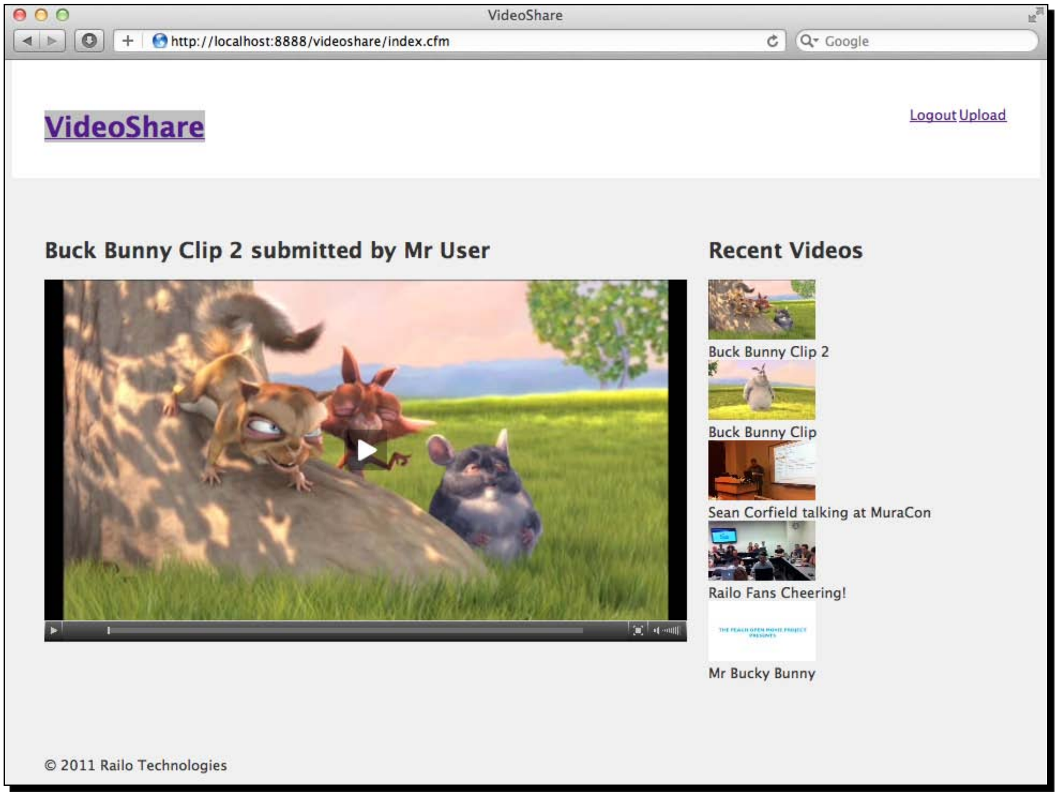
1. Open up `index.cfm` and add the following code to get the latest published videos:

```
<cf_layout>
  <cfset Videos = EntityLoad("Video", {status=1}, "uploadedAt
DESC", {maxResults=10})>

  <cfset HeroVideo = Videos[1]>
<cfoutput>
  <div id="content">
      <h2>#HeroVideo.getTitle()# submitted by #HeroVideo.
getUser().getUserName()#</h2>
      <cfvideoplayer video="videos/#HeroVideo.getFile()#.flv"
width="600" height="338"
                preview="posters/#HeroVideo.getFile()#.jpg">
  </div>

  <div id="sidebar">
    <h2>Recent Videos</h2>
    <ul class="thumbList">
      <cfloop array="#Videos#" index="video">
        <li>
        <a href="play.cfm?id=#video.getID()#"><img
src="thumbs/#video.getFile()#.jpg" border="0"></a><br>
        #video.getTitle()#
        </li>
    </cfloop>
    </ul>
  </div>
</cfoutput>
</cf_layout>
```

2. Not a massive change here; we have seen most of it before. First off, we get an array of the latest videos by calling `EntityLoad("Video", {status=1}, "uploadedAt DESC", {maxResults=10})`, which gets all the videos with `status=1`, ordered by the `uploadedAt` property. Then we set that we are only going to return ten results. We then set our `HeroVideo` (the main video on the page) by getting the first item in the `Videos` array.

As we did in the `play.cfm` page, we display the `<cfvideoplayer>` with our `HeroVideo`. Finally, we simply list all the videos in another `<div>` by looping through the whole array of videos we first obtained. Those videos are actually just displayed as the thumbnails we created, and we link them to the `play.cfm` page!

3. Now, after we upload a few more videos, our home page looks much more interesting, as shown in the following screenshot:



VideoShare
http://localhost:8888/videoshare/index.cfm
Google
VideoShare
Logout Upload
Buck Bunny Clip 2 submitted by Mr User
Recent Videos
Buck Bunny Clip 2
Buck Bunny Clip
Sean Corfield talking at MuraCon
Railo Fans Cheering!
Mr Bucky Bunny
© 2011 Railo Technologies

What just happened?

We finished our application! To get the latest uploaded videos, we used the `EntityLoad("Video", {status=1}, "uploadedAt DESC", {maxResults=10})` function to get all the videos that have a status equal to 1 (which we set after we converted it), we then sorted them by the `uploadedAt` property and then listed them using the thumbnails we created earlier.

Now, our application has all the features we want, and it wasn't that hard to achieve!

Have a go hero

There are areas that we could improve in the application. Why don't you try adding the following features:

- Displaying a User page with his/her information and all the videos they have uploaded
- Maybe use the AJAX capabilities of Railo to make the comments system more interactive
- Display related videos on the video display page by relating videos to each other

Summary

In this chapter, we covered a lot, and hopefully brought together a lot of the features of Railo Server that we learned in other chapters to develop a full-fledged application.

We covered:

- Setting up our `Application.cfc` template so that our application can have access to the ORM capabilities
- Adding an `onSessionStart()` method to the `Application.cfc`—this allowed us to implement a simple security system based on the `SESSION` scope
- Creating components that can easily be persisted in a database—the ORM capability of Railo Server means that you don't have to worry too much about the database schema and just get on with developing your application using components that map to real-world objects
- Converting and getting images from videos— by using the `<cfvideo>` tag. We were able to convert videos easily to a web-displayable format and also generate poster and thumbnails from a video

Hopefully, this chapter has given you an overview of how all the features we learned in previous chapters go together easily. Now, you too can develop complex applications using Railo Server quickly.

Index

[PACKT] open source
PUBLISHING community experience distilled

Thank you for buying
Railo 3 Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

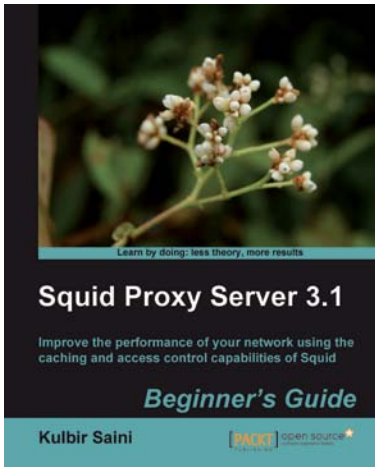
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING


Learn by doing: less theory, more results
Squid Proxy Server 3.1
Improve the performance of your network using the caching and access control capabilities of Squid
Beginner's Guide
Kulbir Saini
[PACKT] open source

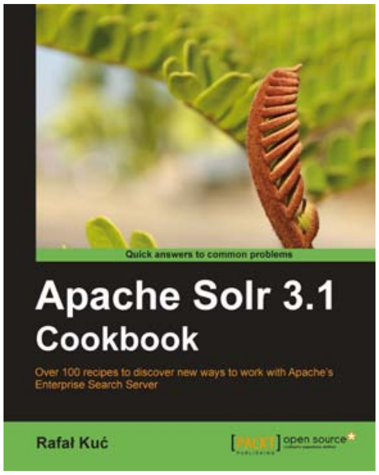
Squid Proxy Server 3.1: Beginner's Guide

ISBN: 978-1-84951-390-6 Paperback: 332 pages

Improve the performance of your network using the caching and access control capabilities of Squid

1. Get the most out of your network connection by customizing Squid's access control lists and helpers
2. Set up and configure Squid to get your website working quicker and more efficiently
3. No previous knowledge of Squid or proxy servers is required
4. Part of Packt's Beginner's Guide series: lots of practical, easy-to-follow examples accompanied by screenshots

---


Quick answers to common problems
Apache Solr 3.1
Cookbook
Over 100 recipes to discover new ways to work with Apache's Enterprise Search Server
Rafał Kuć
[PACKT] open source

Apache Solr 3.1 Cookbook

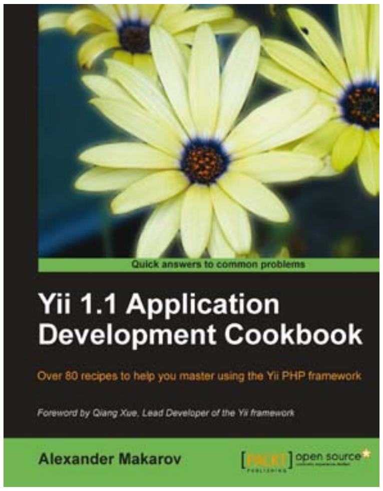
ISBN: 978-1-84951-218-3 Paperback: 300 pages

Over 100 recipes to discover new ways to work with Apache's Enterprise Search Server

1. Improve the way in which you work with Apache Solr to make your search engine quicker and more effective
2. Deal with performance, setup, and configuration problems in no time
3. Discover little-known Solr functionalities and create your own modules to customize Solr to your company's needs
4. Part of Packt's Cookbook series; each chapter covers a different aspect of working with Solr

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
PUBLISHING
community experience distilled


Quick answers to common problems
Yii 1.1 Application Development Cookbook
Over 80 recipes to help you master using the Yii PHP framework
Foreword by Qiang Xue, Lead Developer of the Yii framework
Alexander Makarov
[PACKT] open source

Yii 1.1 Application Development Cookbook

ISBN: 978-1-84951-548-1 Paperback: 392 pages

Over 80 recipes to help you master using the Yii PHP framewor

1. Learn to use Yii more efficiently through plentiful Yii recipes on diverse topics

2. Make the most efficient use of your controller and views and reuse them

3. Automate error tracking and understand the Yii log and stack trace

4. Full of practically useful solutions and concepts that you can use in your application, with clearly explained code and all the necessary screenshots

Community Experience Distilled
Flash
Game Development by Example
Build 9 classic Flash games and learn game development along the way
Emanuele Feronato
[PACKT]

Flash Game Development by Example

ISBN: 978-1-84969-090-4 Paperback: 328 pages

Build 10 classic Flash games and learn game development along the way

1. Build 10 classic games in Flash. Learn the essential skills for Flash game development

2. Start developing games straight away. Build your first game in the first chapter

3. Fun and fast paced. Ideal for readers with no Flash or game programming experience

4. The most popular games in the world are built in Flash.

Please check **www.PacktPub.com** for information on our titles