Learn by doing: less theory, more results

SilverStripe 2.4
Module Extension, Themes, and Widgets

Create smashing SilverStripe applications by extending modules, creating themes, and adding widgets

Beginner's Guide

Philipp Krenn

[PACKT] PUBLISHING open source* community experience distilled

SilverStripe 2.4 Module Extension, Themes, and Widgets

Beginner's Guide

Create smashing SilverStripe applications by extending modules, creating themes, and adding widgets

Philipp Krenn

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

SilverStripe 2.4 Module Extension, Themes, and Widgets

Beginner's Guide

First published: May 2011

Production Reference: 1260411

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849515-00-9

`www.packtpub.com`

Cover Image by Filippo (`Filosarti@tiscali.it`)

Credits

Author
Philipp Krenn

Reviewers
Aaron Carlino
Ingo Schommer
Sigurd Magnusson

Acquisition Editor
Tarun Singh

Development Editor
Meeta Rajani

Technical Editors
Aaron Rosario
Hithesh Uchil

Project Coordinator
Michelle Quadros

Proofreader
Dan McMahon

Indexer
Tejal Daruwale

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Philipp Krenn studies software engineering at the University of Technology, Vienna. At the moment, he is writing his thesis on current database trends. Besides that, he's working as a freelance IT trainer and web developer, mostly using SilverStripe, but also Drupal, CakePHP, and Smarty.

He started using SilverStripe in 2007 as one of the Google Summer of Code students improving the project, beginning with the effort to support multiple databases (besides MySQL). During this, he got a detailed insight into the inner workings of the project. Since then he's been in love with SilverStripe...

Philipp is currently employed at the University of Technology, Vienna as diplomate for an industry project and as IT training manager at Splendit IT Consulting GmbH. When doing freelance work he's frequently working for men on the moon GmbH, on SilverStripe projects, or as IT trainer for SPC GmbH.

Even when working on something inspiring, writing a book is actually hard work—lo and behold ;-).

Therefore a huge THANK YOU to the people making it possible!

Packt Publishing for channeling my enthusiasm into something useful, while helping wherever they could. "Danke" to Michelle Quadros for making our schedule work out, and also Tarun Singh, Meeta Rajani, Hithesh Uchil, and Aaron Rosario for keeping me on the right track.

Ingo Schommer from SilverStripe Ltd. lent more than one helping hand in the creation of this book. Additionally he and my other two reviewers Sigurd Magnusson (SilverStripe Ltd.) and Aaron Carlino (Bluehouse Group) provided crucial input—without you the book would only be half as good.

Last but not least my girl, family, and friends, who lost me for months to writing this book. Without you providing the right background this would have been impossible!

About the Reviewers

Aaron Carlino is a web developer who is better known in the SilverStripe community by his whimsical pseudonym "Uncle Cheese". He has been doing web development since 2005, and has found his niche in SilverStripe programming after an exhaustive search for a content management solution that was welcoming to developers and would stay out of his way. Since then, he has established a strong reputation in the SilverStripe community as a mentor, support provider, and, most notably, a contributor of some of the application's most popular modules including DataObjectManager, ImageGallery, and Uploadify.

During the day, he is employed full-time at Bluehouse Group as lead SilverStripe developer, surrounded by a team of talented designers, programmers, and HTML developers. At Bluehouse Group, he has worked on several sophisticated web applications built on SilverStripe, including All Earth Renewables, ISI, and Yestermorrow. In his spare time, he keeps his SilverStripe thirst quenched by entertaining a variety of freelance projects which range from ad-hoc support work for his modules to full-featured web applications, including Xetic.org. In addition, he has almost always worked on new open-source contributions to the SilverStripe CMS, because, quite frankly, he can't get enough of it.

When he is not coding, he usually thinks about what he'd like to code, and when he's not doing that, he enjoys cooking (and subsequently photographing) all kinds of delicious foods. He is also a talented guitar player and French speaker, as well as a connoisseur of all things Elvis Costello. He lives a blessed and charmed life in beautiful northwestern Vermont with his wife and shih-tzu, Oscar.

Ingo Schommer is a senior developer at SilverStripe Ltd. in Wellington, New Zealand. He is responsible for large scale web application development. Not entirely by chance, he is also a core team member and release manager for the SilverStripe project. Originally hailing from Germany, he was co-author of the first SilverStripe book in his native tongue, quickly followed by an English translation. He's always keen to spread the word about his favorite CMS, and hence thrilled to see a Packt publication on this topic.

Sigurd Magnusson is one of the three co-founders of SilverStripe Ltd (`http://silverstripe.com/`). He has been in this business for more than ten years and currently focuses on sales and marketing. He's been living and breathing the Internet since 1995, when his city council provided the only local internet service, entirely text-based at the time. The potential of the Internet piqued his interest and he began learning computer programming.

While his days at SilverStripe are no longer spent propgramming, he continues to be deeply interested in the technology advances of the Web. Sigurd is an evangelist for the principles and technology of the Web, and is an avid supporter of open source, open data, and the Web as a modern software platform.

He is very familiar with both the commercial and open source segments of the web content management industry, and he influences the direction of the open source SilverStripe CMS and Sapphire framework.

Off the Web, he enjoys spending time with his family, cross-country mountain biking in New Zealand, and experiencing foreign cultures.

Table of Contents

Preface

SilverStripe CMS is an open source web content management system used by governments, businesses, and non-profit organizations around the world. It is a powerful tool for professional web development teams, and web content authors rave about how easy it is to use.

This book is a beginner-friendly introduction to SilverStripe and the only printed documentation for the current 2.4 release. While it starts off nice and easy, we progress fast, covering both SilverStripe's intuitive CMS and powerful framework. We'll show you how to easily extend the core system with your own themes, widgets, and modules by gradually building and extending a graphic example. This unique book helps both frontend designers and backend programmers to make the most of SilverStripe.

What this book covers

Chapter 1, *Principles and Concepts* introduces the software's distinct features, namely being both a framework and a CMS. Additionally, the general approach and structure are described, as well as the ecosystem supporting both developers and users.

Chapter 2, *Customizing the Layout* takes an in-depth look at SilverStripe's template engine. It explains how to build and customize your own layout. The code created here will be extended over the later chapters, providing a simple but already useful introduction. Finally, the chapter teaches how to optimize your search engine ranking with the help of the CMS.

Chapter 3, *Getting "Control" of Your System* explains the underlying architecture of the framework: Model View Controller (MVC). Building on the View role from the previous chapter, this one covers the Controller. Specifically how to create your own page types.

Chapter 4, *Storing and Retrieving Information* explores the Model, the third and final MVC role. SilverStripe provides custom database abstraction, allowing developers to focus on object-oriented code only. Building on this knowledge, you'll learn how to add custom data to the example project.

Chapter 5, *Customizing Your Installation* introduces the most important configuration options. These include settings both for the CMS and the underlying framework, for example configuring the rich-text editor, logging, security, and much more.

Chapter 6, *Adding Some Spice with Widgets and Short Codes* covers SilverStripe's widget system. Specifically it shows how to automatically fetch data from Facebook and how to integrate it into the example project, allowing content editors simply to drag-and-drop content into different pages.

Chapter 7, *Advancing Further with Inheritance and Modules* takes a look at (object-oriented) inheritance and how to make the most of it in combination with modules. A very popular module is used and you'll further extend it, teaching you how to modularize and reuse code.

Chapter 8, *Introducing Forms* makes our site more interactive. It introduces forms and how to easily handle them in SilverStripe. This covers both server and client side validation, how to process inputs and the built-in e-mail capabilities.

Chapter 9, *Taking Forms a Step Further* broadens the concepts from the previous chapter. It adds a general search functionality and then focuses on storing user provided inputs in the database. Additionally validation concepts are explored further.

Chapter 10, *Targeting the Whole World* introduces SilverStripe's powerful globalization features. You'll learn how to take advantage of them in the framework and the CMS, removing language barriers while still keeping it simple.

Chapter 11, *Creating an Application* turns the focus from the public facing website to a fully featured application in the background. Specifically you'll learn how to easily manage data in the CMS and you'll delve deeper into architectural decisions impacting the whole development. **This chapter is available along with the code download for this book.**

Appendix A, *Installing SilverStripe* introduces beginners to SilverStripe's installation process. You'll learn how to install SilverStripe itself, as well as how to set up development and live environments for your sites. **This chapter is available along with the code download for this book.**

Who this book is for

If you are a SilverStripe developer and want to learn the nooks and crannies of developing fully-featured SilverStripe web applications, then this book is for you. Building upon your knowledge of PHP, HTML, and CSS, this book will take you to the next level of SilverStripe development. The book assumes basic experience with SilverStripe.

Conventions

In this book, you will 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: "To include a JavaScript file, instead of plain HTML, you can use `<% require javascript(sapphire/thirdparty/jquery/jquery-packed.js) %>`."

A block of code is set as follows:

```
public static $has_one = array(
  'SideBar' => 'WidgetArea',
);

public function getCMSFields(){
  $fields = parent::getCMSFields();
  $fields->addFieldToTab(
    'Root.Content.Widgets',
    new WidgetAreaEditor('SideBar')
  );
  return $fields;
}
```

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

```
public static $has_one = array(
  'SideBar' => 'WidgetArea',
);

public function getCMSFields(){
  $fields = parent::getCMSFields();
  $fields->addFieldToTab(
    'Root.Content.Widgets',
    new WidgetAreaEditor('SideBar')
  );
  return $fields;
}
```

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

```
pear upgrade --alldeps
```

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: "clicking the **Next** button moves you to the next screen".

[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 for this book

The author mantains an updated repository for this title's code bundle, downloadable from `https://github.com/xeraa/silverstripe-book`. You can also visit `http://www.PacktPub.com/support` for additional support.

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
Principles and Concepts

Whether you have already worked with other content management systems (CMS) or you are a newcomer to this area, SilverStripe's approach and specific feature set will show you something new. It's a flexible and powerful tool for creating both websites and web applications—catering to the requirements of highly customized or out-of-the-box installations.

In this chapter we'll take a short look at:

- Why you should use SilverStripe
- How the basic principles and approach of the system work
- What the general file structure looks like
- What additional resources are available and what to expect from the general ecosystem

As we explore SilverStripe, we will develop a single example project throughout this book. We'll create a website for a bar, enabling the owner to edit various pieces of content himself, including opening hours, prices, and photos. The page will also provide a gallery, contact form, and other interactive parts. Starting with the second chapter, we'll gradually add more and more features and get to know the inner workings of SilverStripe. Later on, we will cover advanced topics such as translating the site, easily managing user data, and sending newsletters.

In case you haven't installed SilverStripe, see the Appendix for detailed instructions. In our example we'll assume that you already have it up and running. Additionally, we won't go into depth explaining the CMS's basics as it's pretty straightforward and well documented. We'll talk about the documentation and support a little later in the chapter.

Why SilverStripe

Whether you were attracted by the "sexy" CMS backend (if you haven't already taken a look at it, you can do so at `http://demo.silverstripe.com`) or the unique set of features, let's take a short look at some of the most notable characteristics. While some parts might not make a lot of sense or seem appealing at first, you're sure to love them later on.

CMS and framework

SilverStripe actually consists of two parts, planned and developed together by the same group of people: the SilverStripe CMS and the Sapphire Framework. Today there are many content management systems and a fair few programming frameworks available. However, SilverStripe's concept of tightly coupling them is a rare but extremely powerful approach. You get the best of two "worlds" and don't need to select one over the other.

This coupling also targets the needs of the two main user groups of the system: developers use the framework to build websites and web applications, whereas content editors rely on the CMS backend.

SilverStripe is a modern system and therefore the code quality is higher than the common perception of PHP products. This also means it can take advantage of newer features in PHP. The benefit of this is that the code you write can be more logical, succinct, and easy to write and change. SilverStripe is not the only modern PHP application in this sense, but this newness does substantially diferentiate it from large, established CMS projects written in PHP and other languages.

Assume you have a simple project with a single developer, Alice and one content editor, Bob.

Alice loves to work with code, and Sapphire enables her to focus on writing HTML, CSS, JavaScript, and (object-oriented) PHP. She is not limited by a graphical interface or the need to study one, but can use the raw power of programming to build complex systems quickly and robustly. In fact, the developer is required to rely on programming in Sapphire as there is no such interface. Although a graphical configuration interface might sound more appealing at first, we'll show you how to achieve more powerful configurations with a few lines of code in our example project. Bob on the other hand doesn't know anything about programming, but there's no need for him to do so. Using the intuitive CMS he can access and edit any content without ever having to leave the graphical interface. This lowers the entry barrier and enables editors like Bob to efficiently work with SilverStripe, only requiring minimal training.

Openness and interoperability

SilverStripe is a strict supporter of the open source movement. It's even seen as one of the main reasons for its success. Source code, updates, documentation, and more are provided free of charge, and everyone is encouraged to actively participate wherever they see fit.

> SilverStripe is released under the Berkeley Software Distribution (BSD) license, a permissive license model originally developed by the University of California, Berkeley. It allows you to copy, modify, and distribute software, even commercially, without having to grant access to the modified source code as long as the original license stays in place. That means you can create your own extensions for paying customers without having to release your changes; most other CMS don't allow this. The BSD license can be found at `http://www.opensource.org/licenses/bsd-license.php`.

Interoperability is another virtue of SilverStripe. Its core technologies (PHP, MySQL, JavaScript, HTML, and CSS) are open standards, widely used, easy to learn and likely to last for many years. A lot of work has gone into supporting all major browsers and operating systems, different web servers and databases, multiple languages and character sets, accessibility, and more.

Getting your job done

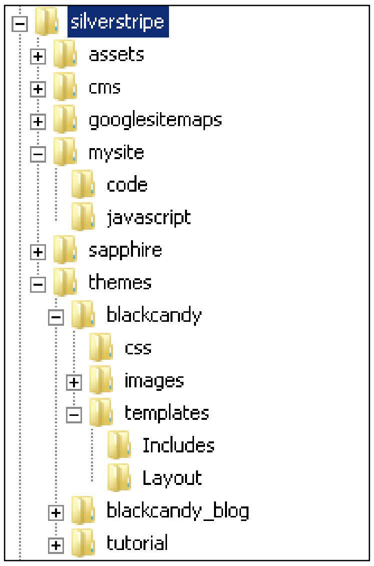
We won't go into too much detail, but here's a brief overview of what developers should expect and how things are done in "SilverStripe land":

- Like in any good web application framework, HTML, PHP, and database queries are not mixed but nicely separated in three layers: one for storing information in the database, one containing the logic, and one responsible for the output. This really helps keep the code clean and easy to maintain.
- The database layer is abstracted so you don't need to care for various quirks of the different databases. MySQL, PostgreSQL, SQLite, and Microsoft SQL Server are fully supported; an Oracle module is currently being worked on and is marked as experimental. Changes to this layer are also synchronized with the database, saving you from making manual changes in various places.
- Customization is achieved through class inheritance, this helps to avoid repetition. Consider our developer Alice and content editor Bob. We could say that both are a "person", having a name, sex, age, and so on. "Developers" additionally have skills like programming languages, while "content editors" don't. Relying on inheritance we could model the entity "person" once and only add specific attributes to the so-called sub classes of "developers" and "content editors" as needed; without having to repeat the basic person information multiple times. This approach is common in programming frameworks, but it's rare for a CMS to support it. The bigger and more complex your system gets, the more this principle will shine as we'll see later on.

- SilverStripe's core can be extended by modules (adding functionality, such as a blog), themes (giving your project a different look), and widgets (including small features, such as a map on a contact page).

The file structure

Let's take a look at the file structure. We'll assume you've just installed the latest version successfully, so you should see the following:



```
silverstripe
  assets
  cms
  googlesitemaps
  mysite
    code
    javascript
  sapphire
  themes
    blackcandy
      css
      images
      templates
        Includes
        Layout
    blackcandy_blog
    tutorial
```

assets/

This folder contains all the files uploaded by website and CMS users. Content editors can access any file under the **Files & Images** menu point in the CMS backend. Keeping all user files in a single place has two advantages:

- Having a clean file structure
- Being able to limit SilverStripe's write permissions to this folder

cms/

All the files located here are used to power the CMS interface. We won't be changing any of these.

googlesitemaps/

This is a module for rendering your page's sitemap in XML, which improves the communication with search engines. We'll talk more about that in the next chapter. If you don't need this feature you could remove the folder to get rid of it.

[Note that modules in SilverStripe are always located in the root folder.]

mysite/

This folder contains our custom PHP and JavaScript code. Its default name is `mysite/`, but you can also rename it, if you want to. We'll take a look at that in chapter three.

Actually, `mysite/` is a module itself, extending the base system according to our requirements. This folder is the core of any customization. Like in any other module, there is a `mysite/_config.php` file, which contains the site's configuration. This includes the database credentials and much more, as we'll see.

sapphire/

This is the heart of your application as it contains the Sapphire framework. The other parts of the system rely on the core classes contained in this folder.

themes/

Although the `mysite/` folder accommodates our custom logic, this folder houses our site's layout—consisting of template, CSS, and image files. This directory can contain multiple theme subfolders. By default, these include the standard theme BlackCandy.

SilverStripe's ecosystem

SilverStripe is a very powerful tool, supported by a whole ecosystem. Let's take a quick look at it so you can make the most of it.

The company behind SilverStripe

While SilverStripe is open source software, the source code is managed and mainly produced by a privately held company, SilverStripe Ltd.. The headquarters are in New Zealand, but there's also a branch in Australia.

This is unusual as most open source projects are developed and maintained by a community without a company in the background. In this case, however, it does not mean that there is a limited, free community edition and a full, commercial one; there is only a single, free version with all the features SilverStripe Ltd. itself uses for its commercial projects.

Based on this model, you actually get the best of two worlds: on the one hand you can freely access the source code, documentation, roadmap, and bug tracker. On the other hand you can easily get commercial support, or the development of whole projects, from the main SilverStripe developers themselves. That's a huge selling point as help is always only a phone call away and you don't have to rely on someone's goodwill to help you. Enterprise-level organizations are likely to require such a support system before even considering making use of specific software.

For more details on the company and the professional services that they provide, visit `http://www.silverstripe.com`. Source code, documentation, and much more is available on `http://www.silverstripe.org`.

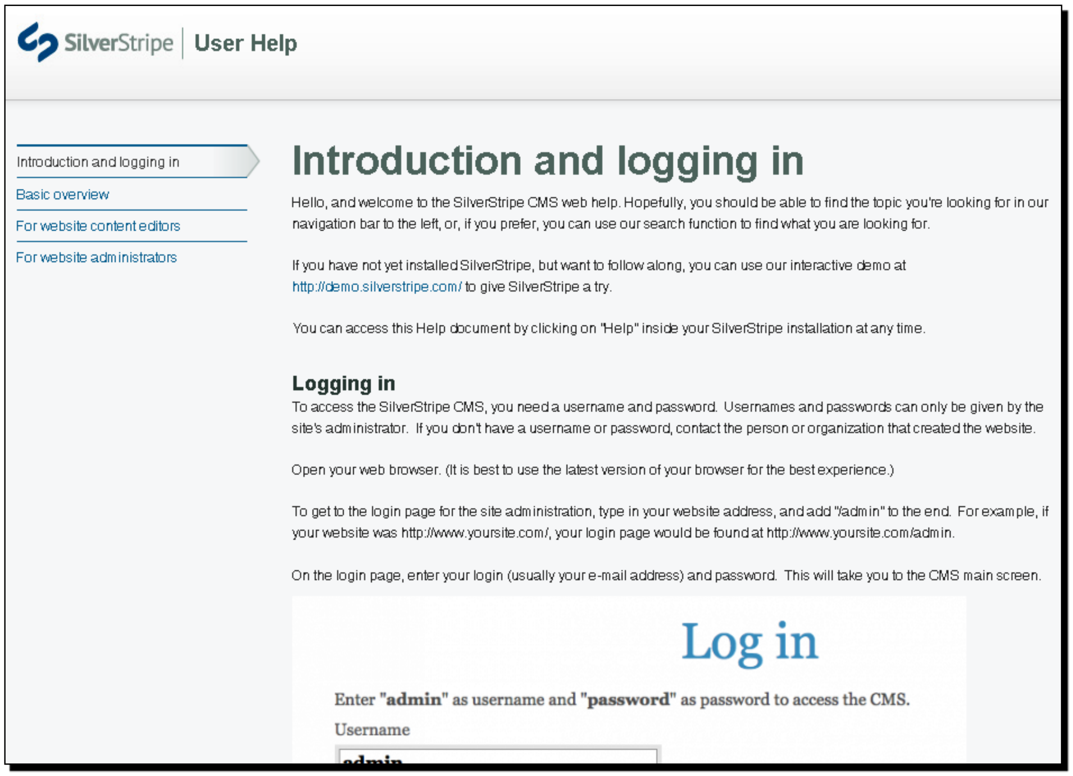
Documentation

SilverStripe provides a vast amount of freely available documentation. While this book will cover many different topics and provide you with all the tools needed to accomplish common tasks, it can never include every little detail. And as the development never stops, some features of SilverStripe might change, but such changes will be included in the documentation.

For an overview of all the available resources, head over to `http://www.silverstripe.org/help-and-support/`.

User help

In order to get to know the nuts and bolts of the CMS take a look at `http://userhelp.silverstripe.org`. We'll not cover that part in the book as it is very well documented and mainly self-explanatory. It is also available from within the CMS through the **Help** button.


SilverStripe | User Help
Introduction and logging in
Basic overview
For website content editors
For website administrators

Introduction and logging in

Hello, and welcome to the SilverStripe CMS web help. Hopefully, you should be able to find the topic you're looking for in our navigation bar to the left, or, if you prefer, you can use our search function to find what you are looking for.

If you have not yet installed SilverStripe, but want to follow along, you can use our interactive demo at http://demo.silverstripe.com/ to give SilverStripe a try.

You can access this Help document by clicking on "Help" inside your SilverStripe installation at any time.

Logging in

To access the SilverStripe CMS, you need a username and password. Usernames and passwords can only be given by the site's administrator. If you don't have a username or password, contact the person or organization that created the website.

Open your web browser. (It is best to use the latest version of your browser for the best experience.)

To get to the login page for the site administration, type in your website address, and add "/admin" to the end. For example, if your website was http://www.yoursite.com/, your login page would be found at http://www.yoursite.com/admin.

On the login page, enter your login (usually your e-mail address) and password. This will take you to the CMS main screen.

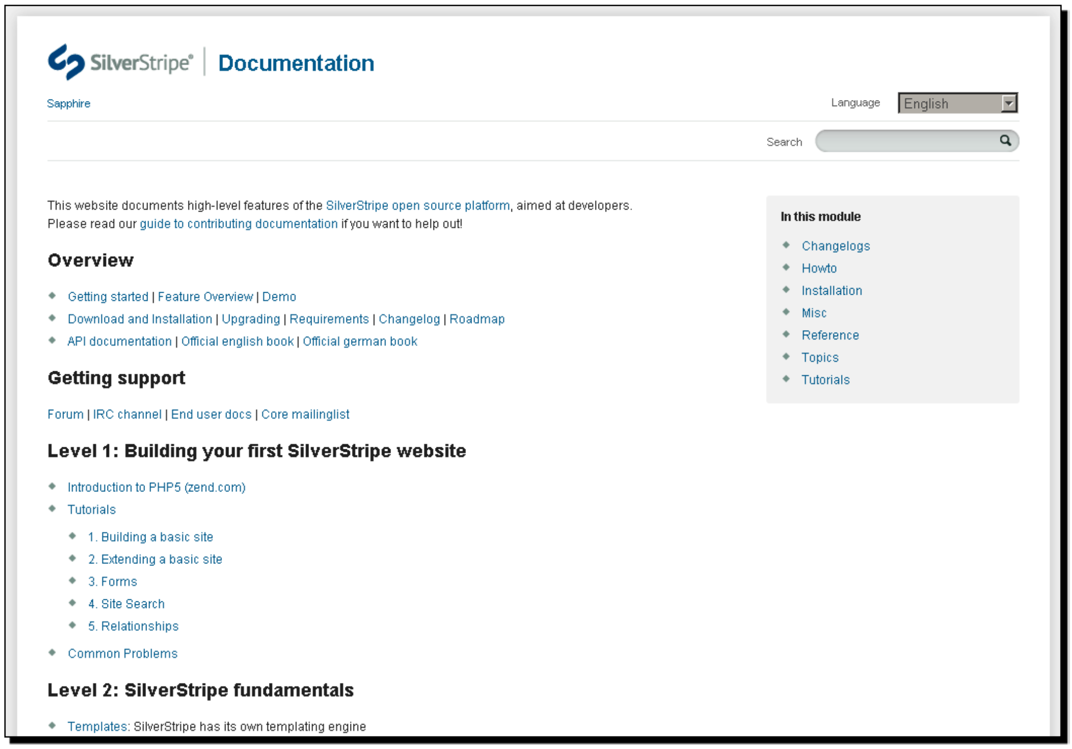
Log in

Enter "**admin**" as username and "**password**" as password to access the CMS.

Username

Developer documentation wiki

`http://doc.silverstripe.org` holds the official documentation, including information on how to install SilverStripe, how to get started, and some more advanced topics. Some articles are more comprehensive than others. This book will cover all the basics and most of the details documented there.



This website documents high-level features of the SilverStripe open source platform, aimed at developers.
Please read our guide to contributing documentation if you want to help out!

Overview

- Getting started | Feature Overview | Demo
- Download and Installation | Upgrading | Requirements | Changelog | Roadmap
- API documentation | Official english book | Official german book

Getting support

Forum | IRC channel | End user docs | Core mailinglist

Level 1: Building your first SilverStripe website

- Introduction to PHP5 (zend.com)
- Tutorials
  - 1. Building a basic site
  - 2. Extending a basic site
  - 3. Forms
  - 4. Site Search
  - 5. Relationships
- Common Problems

Level 2: SilverStripe fundamentals

- Templates: SilverStripe has its own templating engine

In this module

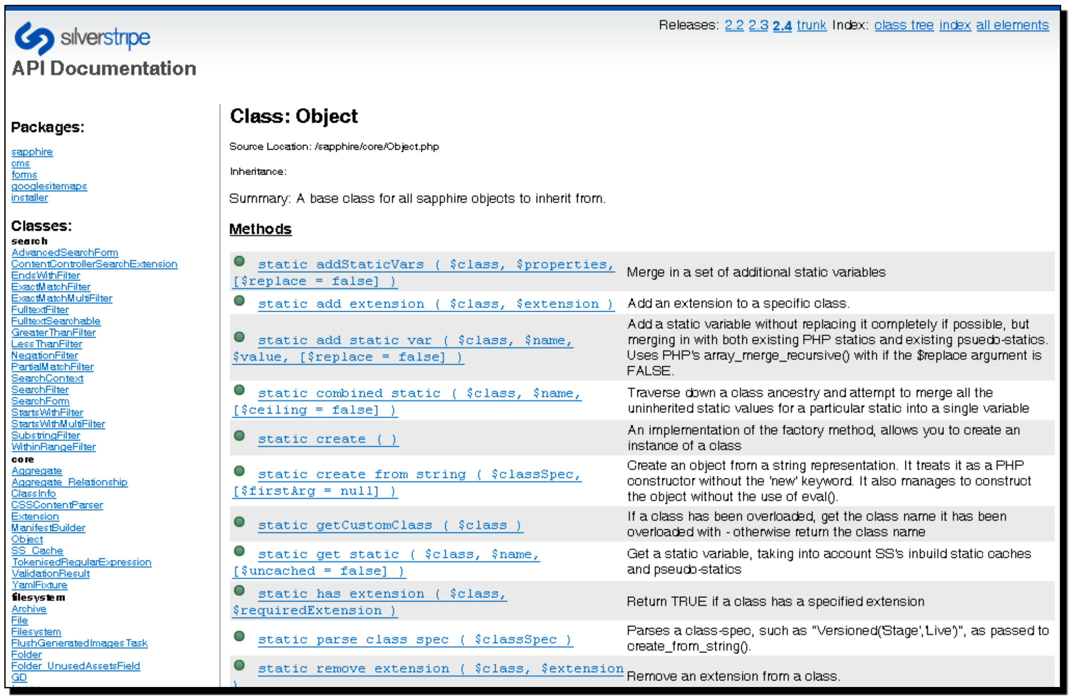
- Changelogs
- Howto
- Installation
- Misc
- Reference
- Topics
- Tutorials

API documentation

If you are an advanced user, which you will definitely be by the end of this book, you'll be most interested in the **Application Programming Interface** (**API**): `http://api.silverstripe.org`.

While the documentation covers specific tasks step by step, the API describes how specific functions can be called, how they interact, and how to use them in a concise and technical fashion.

The API covers all the available parts of Sapphire, the SilverStripe CMS, and modules created by the core developers.



silverstripe
API Documentation

Releases: 2.2 2.3 **2.4** trunk Index: class tree index all elements

Packages:

sapphire
cms
forms
googlesitemaps
installer

Classes:

search
AdvancedSearchForm
ContentControllerSearchExtension
EndsWithFilter
ExactMatchFilter
ExactMatchMultiFilter
FulltextFilter
FulltextSearchable
GreaterThanFilter
LessThanFilter
NegationFilter
PartialMatchFilter
SearchContext
SearchFilter
SearchForm
StartsWithFilter
StartsWithMultiFilter
SubstringFilter
WithinRangeFilter
core
Aggregate
Aggregate_Relationship
ClassInfo
CSSContentParser
Extension
ManifestBuilder
Object
SS_Cache
TokenisedRegularExpression
ValidationResult
YamlFixture
filesystem
Archive
File
Filesystem
FlushGeneratedImagesTask
Folder
Folder_UnusedAssetsField
GD

Class: Object

Source Location: /sapphire/core/Object.php

Inheritance:

Summary: A base class for all sapphire objects to inherit from.

Methods

● `static addStaticVars ( $class, $properties, [$replace = false] )`	Merge in a set of additional static variables
● `static add extension ( $class, $extension )`	Add an extension to a specific class.
● `static add static var ( $class, $name, $value, [$replace = false] )`	Add a static variable without replacing it completely if possible, but merging in with both existing PHP statics and existing psuedo-statics. Uses PHP's array_merge_recursive() with if the $replace argument is FALSE.
● `static combined static ( $class, $name, [$ceiling = false] )`	Traverse down a class ancestry and attempt to merge all the uninherited static values for a particular static into a single variable
● `static create ( )`	An implementation of the factory method, allows you to create an instance of a class
● `static create from string ( $classSpec, [$firstArg = null] )`	Create an object from a string representation. It treats it as a PHP constructor without the 'new' keyword. It also manages to construct the object without the use of eval().
● `static getCustomClass ( $class )`	If a class has been overloaded, get the class name it has been overloaded with - otherwise return the class name
● `static get static ( $class, $name, [$uncached = false] )`	Get a static variable, taking into account SS's inbuild static caches and pseudo-statics
● `static has extension ( $class, $requiredExtension )`	Return TRUE if a class has a specified extension
● `static parse class spec ( $classSpec )`	Parses a class-spec, such as "Versioned('Stage','Live')", as passed to create_from_string().
● `static remove extension ( $class, $extension )`	Remove an extension from a class.

Community

If you're stuck with a problem or need some advice on how to accomplish a task, don't hesitate to get in touch with the community (for example, if you have problems installing the CMS). It's an ever-increasing pool of like-minded people, actively discussing problems and helping each other. It consists of four main communication channels, detailed below.

Forum

`http://www.silverstripe.org/forums/` is the official SilverStripe forum, which is a good place to ask general questions as well as get help with specific problems. The response time ranges from minutes to hours, depending on the time of day you post a question. The core developers visit the forum frequently, so you can generally expect helpful responses by them or other community members. Additionally, when you sign up to the forum, you can subscribe to a monthly SilverStripe community newsletter.

IRC chat

IRC is the quickest way of getting help. Depending on the time of day you might have just a few or up to dozens of communication partners. It's especially useful for asking rather simple and short questions and the core developers are also frequently available.

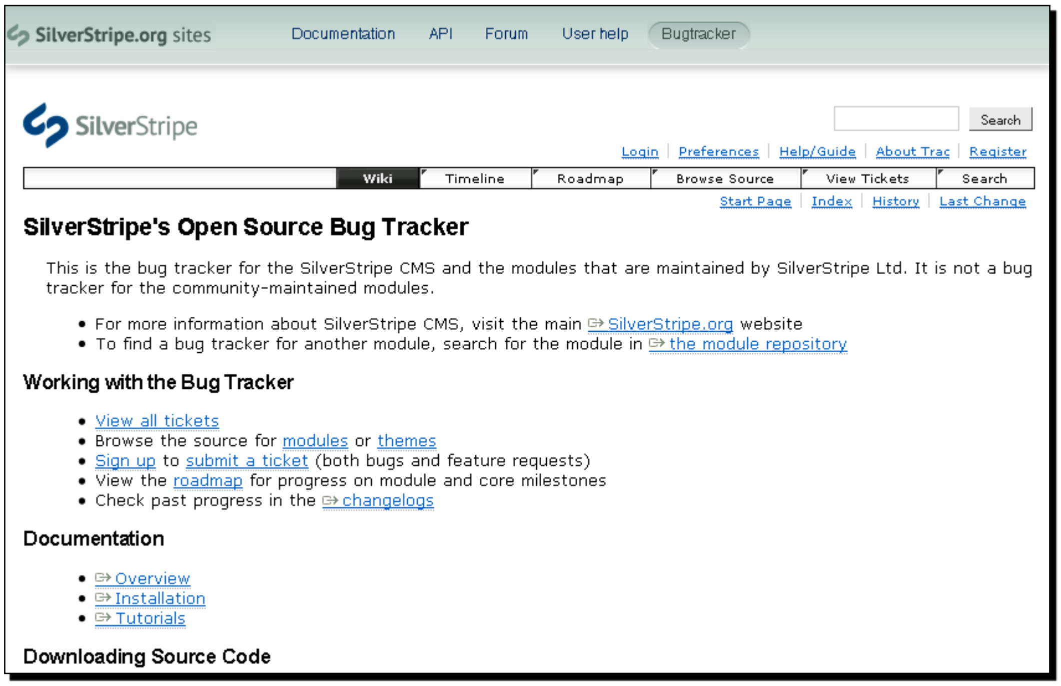
The server used is `irc.freenode.net` and the channel `#silverstripe`—for more details, web based access, or logs, head over to `http://www.silverstripe.org/irc-channel/`.

Development mailing list

`http://groups.google.com/group/silverstripe-dev/` is the place to discuss the general direction and decisions concerning SilverStripe's development. It's a good place to find out what's new and currently under development.

Bug tracker

Here you can see the roadmap of the project, review the changelogs, or report bugs you have found: `http://open.silverstripe.org`



SilverStripe's Open Source Bug Tracker

This is the bug tracker for the SilverStripe CMS and the modules that are maintained by SilverStripe Ltd. It is not a bug tracker for the community-maintained modules.

- For more information about SilverStripe CMS, visit the main SilverStripe.org website
- To find a bug tracker for another module, search for the module in the module repository

Working with the Bug Tracker

- View all tickets
- Browse the source for modules or themes
- Sign up to submit a ticket (both bugs and feature requests)
- View the roadmap for progress on module and core milestones
- Check past progress in the changelogs

Documentation

- Overview
- Installation
- Tutorials

Downloading Source Code

Twitter

If you're a Twitter user, you can follow `@silverstripe` for the latest updates. The commonly-used hashtag is `#silverstripe`.

Summary

We've already learned quite a lot about the general principles of SilverStripe and its ecosystem, specifically:

- Why SilverStripe is unique and distinct from other CMSs and frameworks
- Which basic principles are used throughout the system
- Which folders are available in general and what they are used for
- How SilverStripe is developed and why it's useful to have a company behind an open source project
- What resources are available, where to find documentation, and how to get help

In the next chapter, we'll build upon the general concepts we've learned so far by exploring SilverStripe's template engine. This will enable you to create and style the interface of our example project.

2
Customizing the Layout

In this chapter we'll focus on the layout, which is the ideal candidate for starting off with, as it is the most visible part of any project. We assume a good understanding of HTML/XHTML and CSS as well as a basic knowledge of JavaScript and image editing. For the following chapters, a solid understanding of PHP with a focus on object-orientation is also highly recommended.

At first we'll explore SilverStripe's standard theme BlackCandy. After that we shall create the basic layout for our bar website, which we'll extend and enrich over the coming chapters.

Technically speaking, we'll cover the following goals:

- Explore what themes are
- Learn how to use SilverStripe's template engine
- Discover how to edit the layout of our pages, both on the final page and in the CMS backend
- Get an overview of available controls and how to use them
- Build our own menus
- Package our customizations into our own theme
- Learn how to debug errors in the template
- Explore how to improve our search engine ranking with SilverStripe

If you want to improve on your PHP and object-orientation coding, you might want to take a look at Hasin Hayder's book *Object-Oriented Programming with PHP5* (published by Packt Publishing, ISBN 1847192564).

Templates and themes

All pages within SilverStripe are rendered using a template, which is basically an HTML/XHTML file with added control code. A template allows you to make conditional statements, create menus, access data stored in the database, and much more. Instead of using PHP for these enhancements, templates rely on a dedicated template engine. When the user accesses a page the engine replaces the control code with PHP. This transformation is then evaluated by the web server to regular markup code, which every browser can display.

A theme consists of a collection of template, CSS, and image files. Design-specific JavaScript is also stored here. Together they form the layout of a page.

Switching between themes

You can have multiple themes and switch between them whenever you like.

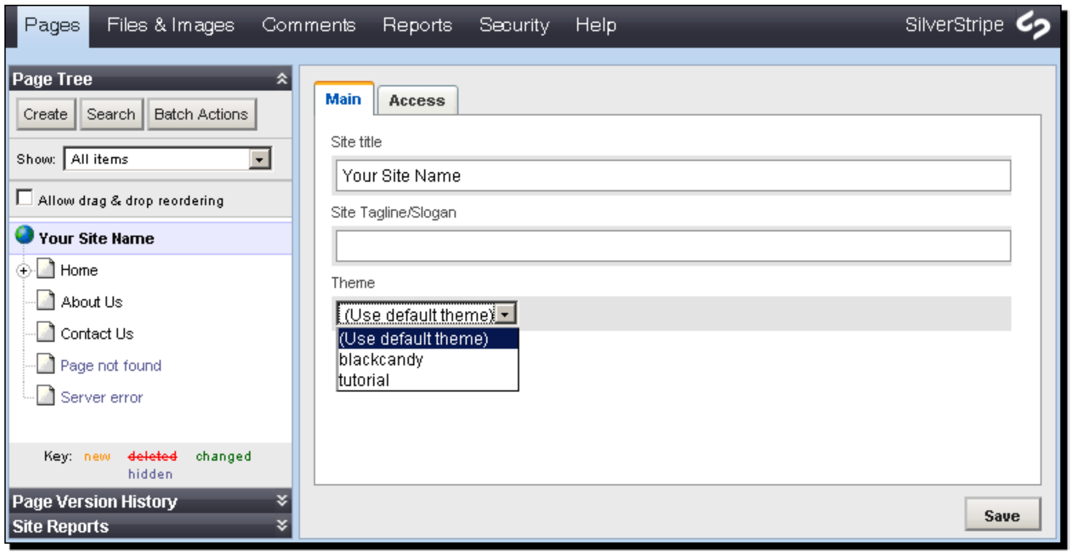
In case you haven't yet installed SilverStripe, you can find detailed instructions in the Appendix. You're definitely encouraged to give everything described here a try. It helps you in becoming an active developer instead of a passive user. So let's get things started!

Time for action – change the default theme

Simply perform the following steps in the CMS and you're all set:

1. Log into the CMS by appending `/admin` to your page's base URL. Provide the credentials you defined during the installation and you should be logged-in.
2. Go to the general configuration page by clicking on the globe within the **Page Tree** section. The default text of this element is **Your Site Name**.

3. The **Theme** is set to **(Use default theme)** by default; change this to **tutorial**.



Site title: Your Site Name

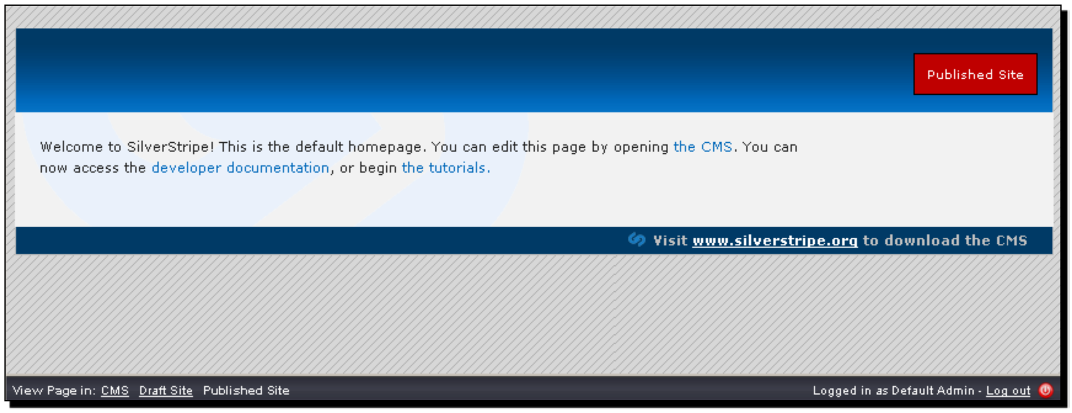
Site Tagline/Slogan:

Theme: (Use default theme)

(Use default theme)
blackcandy
tutorial

4. Click on the **Save** button.

5. Go to the base URL / and reload the page.


Published Site
Visit www.silverstripe.org to download the CMS
View Page in: CMS Draft Site Published Site
Logged in as Default Admin - Log out

Welcome to SilverStripe! This is the default homepage. You can edit this page by opening the CMS. You can now access the developer documentation, or begin the tutorials.

What just happened?

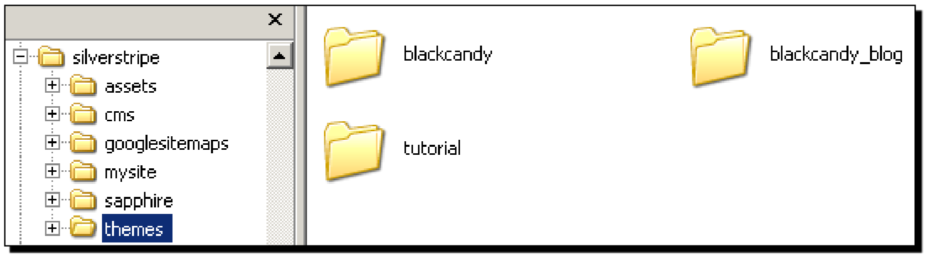
We've successfully changed the theme of our website. It's as easy as that.

Note that you need to package your theme into a subfolder of `themes/` to be able to switch between them.

While you can also put the contents of a theme directly into the `mysite/` folder, keeping your system modularized is a good practice. Therefore we won't integrate the layout into the site specific folder and will instead always create a dedicated, switchable theme.

Getting more themes

There are many prebuilt themes that you can download at `http://silverstripe.org/themes/`. After unpacking a theme, copy it to the `themes/` folder, change the theme in the CMS, and you're good to go.



Using a prebuilt theme really is that simple. Next, we'll take a look at the facilities for building our own themes—specifically, the template engine that powers all themes in SilverStripe.

By the end of this chapter, you'll even be ready to upload and share your own themes with the rest of the community at the address above. New contributions are always welcome.

Template engine

As described earlier, the SilverStripe architecture consists of three layers, each serving a specific purpose. The one responsible for the layout is called **View**, which we'll cover in detail in this chapter. The other two, namely Controller and Model, will be covered later on.

Another template engine?

One might wonder why it is necessary to learn another template language, while there are already so many others available and PHP can do all of it as well. The main reasons behind this are:

- The template engine and the rest of the system fit perfectly together. After getting the hang of it, it makes the creation of templates and communication with the other layers faster and easier.

- The available control code is very simple. Designers without a lot of programming knowledge can create feature-rich layouts.
- It enforces good coding practice. Without the ability to use raw PHP, one cannot by-pass the other layers and undermine the architecture. Only layout and rendering information should be used within templates.

Taking a look at BlackCandy

First of all, switch back to the BlackCandy theme as we want to take a better look at it.

Within your installation, navigate to the folder `themes/blackcandy/` and you'll see three folders: `css/`, `images/`, and `templates/`. The `images/` folder contains any image used in your theme; if you like, you can create subfolders to keep things organized. The remaining folders are a bit more complicated, so let's break them down.

CSS

At first this looks a bit messy—five files for the CSS; couldn't we use just a single one? We could, but many developers consider it a good practise splitting the functionality into different parts, making it easier to navigate later on. Sticking to this convention adds functionality as well.

This is a common principle, called **convention over configuration.** If you do things the way they are expected by the system, you don't need to configure them specifically. Once you know the "way" of the system, you'll work much quicker than if you had to configure the same things over and over again.

editor.css

This file is automagically loaded by SilverStripe's **what you see is what you get** (**WYSIWYG**) editor. So if you apply the correct styling through this file, the content in the CMS' backend will look like the final output in the frontend. Additionally you'll have all custom elements available under the **Styles** dropdown, so the content editors don't need to mess around with pure HTML.

As this file is not automatically linked to the frontend, it's common practice to put all frontend styling information into `typography.css` and reference that file in `editor.css`:

```
@import "typography.css";
```

If you want to provide any styling information just for the CMS, you can put it below the `@import`.

layout.css, form.css, and typography.css

These files are automagically included in the frontend if they are available in your theme. While `layout.css` is used for setting the page's basic sections and layout, `form.css` deals with the form related styling.

`typography.css` covers the layout of content entered into the CMS, generally being imported by `editor.css` as we've just discussed. Elements you will include here are headers (`<h1>`, `<h2>`, and so on), text (for example `<p>`), lists, tables, and others you want to use in the CMS (aligning elements, `<hr>`, and so on).

ie6.css

This file isn't part of the SilverStripe convention, but is a useful idea you might want to adopt. You must include it manually but it will still be a good idea to stick to this naming schema: `ie.css` for styling elements in any version of Internet Explorer, `ie6.css` for Internet Explorer 6 specifics, and so on.

What about performance?

Cutting down on the number of files being loaded is an effective optimization technique for websites. We'll take a look at how to do that and still stick to the naming convention in the next chapter.

Templates

Now that we've discussed the styling, let's take a look at how to put the content together with the help of templates.

Learning the very basics

Before we continue, some basic facts and features of templates:

- Templates must use the file extension `.ss` instead of `.html` or `.php`.
- Templates can be organized alongside their PHP Controllers, so you can use the same powerful mechanism as in the other layers. We'll take a better look at what this means a little later.
- Page controls consist of placeholders and control structures, which are both placed within the regular markup language.
- Placeholders start with a `$` and are processed and replaced by the template engine.
- Control structures are written between opening `<%` and closing `%>`. They look a bit like HTML tags and they are used the same way. Some consist of a single element, like HTML's `<br>`, whereas others consist of two or more.

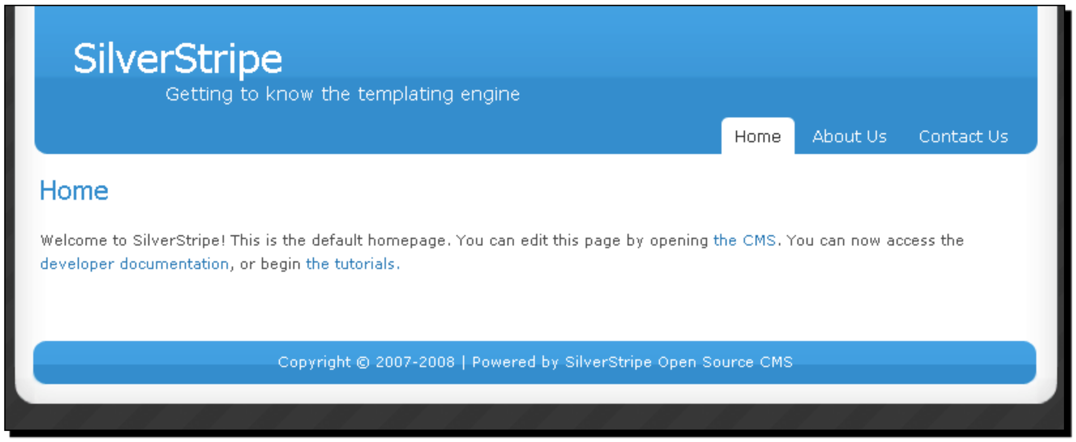
Starting to use templates

Now that we've covered the basics, let's put them into practice.

Time for action – using site title and slogan

We shall use placeholders, taking a look at the code level and how to use them in the CMS:

1. Open the file `themes/blackcandy/templates/Page.ss` in your preferred editor.
2. If the syntax highlighting is disabled due to the unknown file extension, set it to HTML.
3. Find the two placeholders `$SiteConfig.Title` and `$SiteConfig.Tagline` in the code.
4. Go to the general configuration page in the CMS, the one where we've changed the theme.
5. Edit the **Site title** and **Site Tagline/Slogan** to something more meaningful.
6. **Save** the page.
7. Go to the base URL and hit **refresh**.
8. If you view the page's source, you can see that the two placeholders have been replaced by the text we have just entered in the CMS.



Home

Welcome to SilverStripe! This is the default homepage. You can edit this page by opening the CMS. You can now access the developer documentation, or begin the tutorials.

Copyright © 2007-2008 | Powered by SilverStripe Open Source CMS

If you are wondering about the short doctype declaration: there is nothing missing, this is HTML5, which SilverStripe is already using in its default theme. If you prefer XHTML or an older HTML standard, you can freely change it. The CMS happily works with all of them.

What just happened?

The file we've just edited is the base template. It's used for every page (unless specifically overwritten). You can define your general layout once and don't have to repeat it again.

You should always try to avoid repeating code or content. This is generally called **don't repeat yourself** (**DRY**) or **duplication is evil** (**DIE**). SilverStripe supports you very well in doing this.

The site title and slogan are globally available. You can use them on every page and they share the same content across the whole website. These placeholders are prefixed with `SiteConfig.` as well as a `$` sign. In the CMS they are all available on the site's root element, the one with the globe. By default, there are just two, but we'll later see how to add more.

Other placeholders are available on specific pages or can be different on each page. We'll come to those next.

Layout

Looking again at the `Page.ss` we've just opened, you'll also see a `$Layout` placeholder. This is replaced by a file from the `themes/blackcandy/templates/Layout` folder. There are only two files available by default and for every standard page `Page.ss` is used.

When a page is loaded, the template engine first finds the base `templates/Page.ss` (excluding the theme specific part of the path as this can vary). It's evaluated and the `$Layout` is replaced by `templates/Layout/Page.ss`, which is also evaluated for further SilverStripe controls.

Ignore `Page_results.ss` for the moment. It's only used for search queries which we'll cover later. We'll also add more page types so the layout's `Page.ss` can then be replaced by a more specific template, while the base `templates/Page.ss` is always used.

Includes

Both `Page.ss` files include statements like `<% include BreadCrumbs %>`. These controls are replaced by files from the `themes/blackcandy/templates/Includes/` folder. For example, the above include grabs the `themes/blackcandy/templates/Includes/BreadCrumbs.ss` file.

Note that filenames are case sensitive. Otherwise you're free to select a meaningful name. Try sticking to one naming convention to make your own life easier later on, and also note that you mustn't include the `.ss` extension.

If you're seeing a **Filename cannot be empty** error, make sure the included file really exists.

Have a go hero – using page name, navigation label, and metadata title

Now that we've explored all available files and how they are used, let's take a better look at the system.

In the CMS backend, go to the **Home** page. Change the text currently entered into **Page name**, **Navigation label** (both in the **Main** tab), and **Title** (**Metadata** tab). Take a look at:

- Where on the page they are used
- In which file they are located (take a look at all three possible locations)
- What template placeholders represent them

You might notice `$MetaTitle`, `$Title`, and `$MenuTitle` in your template files (for the moment ignore the appended `.XML`). We will explore these later on.

Page control overview

We've already covered some of the page controls, but let's look at a few more in detail, so you can take full control over the look of your pages.

The syntax allows the following three use cases:

- `$Placeholder` simply calls the placeholder.
- `$Placeholder(Parameter)` adds a parameter to the placeholder call. Think of it as a function or method call: `Placeholder("Parameter")`.

- `$Placeholder.Subelement` or `$Placeholder.Cast` follows the same syntax. The first one accesses a sub-element of the given placeholder, specifically the property of an object or an array element. The second one casts or formats its placeholder, for example by defining a specific date format.

More placeholders

So far we've taken a look at the site-wide placeholders, `$Layout`, and the three different title elements on each page. You might already have noticed some more in the templates. Let's cover the most common ones:

`$BaseHref`	Gets the part of the URL that is common to all pages. This is either the current domain or the subfolder that contains your SilverStripe installation. For example, if you're on the page `http://silverstripe.org/help-and-support/` it will be `http://silverstripe.org`. This can be handy if you want to define the base address in the HTML's `head` section: `<base href="$BaseHref"></base>`
`$Breadcrumbs`	Includes the so-called breadcrumb. In BlackCandy it is wrapped within an include. On a **Subpage** of **Home** it will be **Home | Subpage** where all pages except for the last one are links.
`$Content`	Is replaced by the CMS' main **Content** area of each page.
`$Created`	Time and date when the specific page was created in the CMS.
`$Form`	Most default forms replace `$Form`, for example, the admin area's login. But there are also exceptions like `$SearchForm` below.
`$LastEdited`	Time and date when the specific page was last edited in the CMS. This can be handy to show visitors how up-to-date your content (hopefully) is.
`$Link`	Provides a link to the specific page. This can be used to provide the links in a menu: `<li><a href="$Link">$MenuTitle</a></li>`

`$LinkingMode`	This placeholder and the two below behave very similarly. All three aid you in the styling of menus when used as HTML class attributes. People expect that menu items change color when you select them; this placeholder helps you to achieve this. Depending which page you're on, this one will return `current`, `link`, or `section`. `current`: Returned if you currently are on this page. `link`: Returned if you aren't on this page or one of its sub pages. `section`: Returned if you're currently in this section of the page, meaning one of its subpages is being displayed. Taking a **Subpage** of **Home** as an example: `$LinkingMode` gets replaced by `current` on the **Subpage**, `section` on **Home**, and `link` on any other page. You can use it like this: `<a href="$Link" class="$LinkingMode">$Title</a>` in your template.
`$LinkOrCurrent`	This one doesn't use `section`; it's replaced by `link` , therefore having only two possible values `link` and `current`.
`$LinkOrSection`	Returns either `link` or `section`; `current` is replaced by `section`.
`$MetaTags`	This outputs meta-tags based on your input in the CMS, namely title, keywords, and description. If you want to use some custom logic for the title (like in the BlackCandy theme), you can exclude this tag by setting `$MetaTags(false)`.
`$PageComments`	If page comments are enabled on a page (you can do this in the CMS backend), they are placed here.
`$SearchForm`	By using this placeholder, you can easily include a search form. We'll come back to that later.
`$URLSegment`	Returns the current page's URL, the one you can set in the CMS. For example, if you are on `http://silverstripe.org/help-and-support/` it will return `help-and-support`, on `http://silverstripe.org/help-and-support/more-help/` `more-help`, and on `http://silverstripe.org/help-and-support/?whatever=1` `help-and-support`. So it doesn't include URL parameters and possibly isn't unique, as only the last segment is used. This can be useful for setting up page-specific layouts if you use it as a class attribute.

Pop quiz – placeholders

Adding `$Content` or `$Form` to a page is easy. But are we already able to build a dynamic menu, including all the pages we've created in the CMS?

1. Yes, using `$URLSegment` we can do that.
2. Yes, that's what `$LinkingMode` and `$Link` are for.
3. Yes, `$Breadcrumbs` will do that for us.
4. No, so far we're only able to build static HTML menus.

Although we've already covered the placeholders for building dynamic menus, we can't build one as we don't have the tools (yet) to loop over the pages. So far our placeholders only fetch the content from the page that we're currently viewing. For the moment we're stuck with plain old HTML. Before changing that, we'll dive a little deeper into another feature of placeholders.

Casting placeholders

Remember the mysterious `.XML` we've already seen in the templates? It is used to **cast** the placeholder it's appended to.

Note that the dot (`.`) does not necessarily mean casting. It can also be used for a selection within a bulk of information—remember `$SiteConfig.Title` for example.

The two main areas for using placeholder casting are security and formatting.

Security

Without going into too much detail, depending on where you want to use the output, you'll need to encode it differently in order to display it properly and securely.

`.ATT`	Data can be safely inserted into an HTML or XML attribute: All tags and entities are escaped. `<h1>Test ' & "</h1>` becomes `&lt;h1&gt;Test &#39; &amp; &quot;&lt;/h1&gt;`
`.JS`	Data can be safely inserted into JavaScript: Tags and entities stay as they are, quotes are escaped. `<h1>Test ' & "</h1>` becomes `<h1>Test \' & \"</h1>`
`.RAW`	Everything stays as it is, nothing is escaped. `<h1>Test ' & "</h1>` becomes `<h1>Test ' & "</h1>`
`.XML`	Data can be safely inserted anywhere into HTML or XML: All tags and entities are escaped. `<h1>Test ' & "</h1>` becomes `&lt;h1&gt;Test &#39; &amp; &quot;&lt;/h1&gt;`

Using `$Title.XML` we ensure that `$Title` can't contain any raw HTML as it's escaped to simple text.

Date formatting

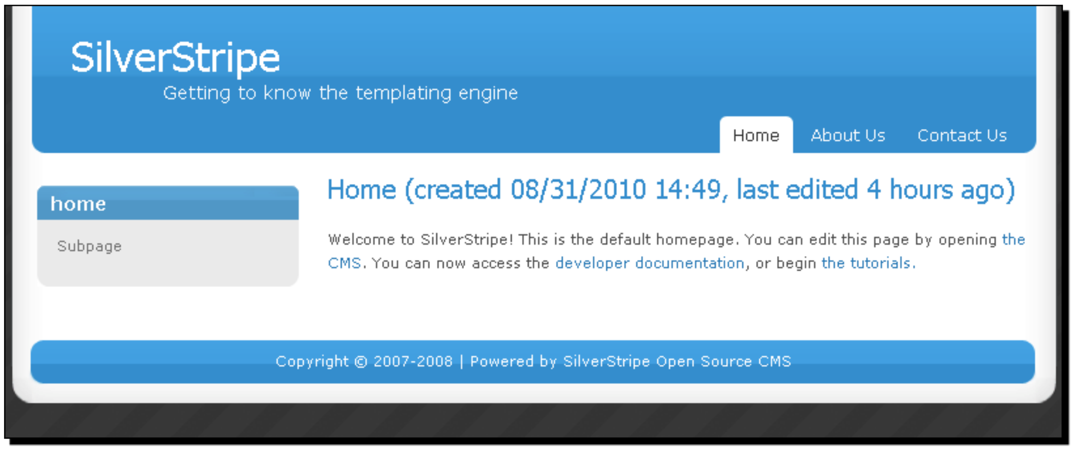
Formatting is common with dates and times, so we'll take a short look at some useful functions for them:

`.Ago`	The seconds, minutes, hours, days, or months (whichever is appropriate) of how long ago the date attribute is: `4 hours ago`
`.Day`	The full day: `Monday`
`.DayOfMonth`	Day of the month: `1`
`.Format(    )`	Provide your own date format, based on the PHP `date()` function for date, time, or both: argument of `d M Y` will result in `01 Jan 2010`.
`.FormatI18N(    )`	Internationalized date, based on the PHP `date()` function: argument of `%B %Y` will result in `Jänner 2010` if you have set your localization to German (more on localization later on).
`.Full`	Full date but with abbreviated month: `1 Jan 2010`
`.InFuture` / `.InPast` / `.IsToday`	Relative time checks. Returning `true` if the condition is met, otherwise an empty string.
`.Long`	Full date: `1 January 2010`
`.Nice`	Format of dd/mm/yy and time in 12-hour format: `20/01/2010 10:00am`
`.NiceUS`	Format of mm/dd/yyyy: `01/20/2010`
`.ShortMonth`	Abbreviated month: `Jan`
`.Time`	Time in 12-hour format: `2:00pm`
`.Time24`	Time in 24-hour format: `14:00`
`.Year`	Year in four-digit format: `2010`

Have a go hero – add the time of the creation and when it was edited

For each page's `title` element add when the page was created and how long ago the last update happened—recall `$Created` and `$LastEdited`. You only need to edit one of the `Page.ss` files—simply find the `$Title` placeholder.

The result should look something like the following image:



Home (created 08/31/2010 14:49, last edited 4 hours ago)

Welcome to SilverStripe! This is the default homepage. You can edit this page by opening the CMS. You can now access the developer documentation, or begin the tutorials.

Copyright © 2007-2008 | Powered by SilverStripe Open Source CMS

Note that you'll need to use the `$Created` placeholder twice to output the data and time exactly like this (once for the date and once for the time—refer to the previous table).

Flushing the cache

Changes to your theme's `templates/Includes/` or `templates/` directories require you to manually flush the cache as these modifications are not automatically picked up; templates are cached for quicker load time. `templates/Layout/` works without it. Flushing is done by appending `?flush=all` to a page's URL. Don't forget this as it is one of the main sources of frustration for beginners when their changes don't seem to have any effect!

Users

Tracking users is a common task for websites. SilverStripe provides three placeholders for making it very easy to do that:

- `$PastVisitor`: If the user has visited the site sometime in the past, it returns `true`, otherwise nothing. The tracking is done via cookies which are automatically set by the CMS.
- `$PastMember`: If the user is a registered member of the site, it returns `true`, otherwise nothing.
- `$CurrentMember`: Checks if the user is currently logged in, returning `true` again. But it can also do much more, providing access to the user's data, just like `$SiteConfig`. For example, `$CurrentMember.FirstName` will output the given name of the currently logged in user.

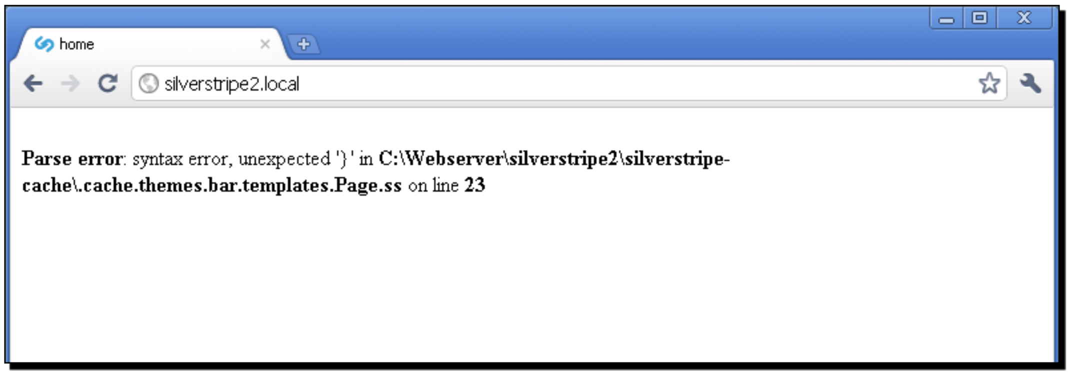
If your site suddenly addresses you as **Default Admin**, you are using the default admin account and haven't yet set your name. You can do that by going to the page `/admin/security/` in your installation.

Control structures

Now that we've covered placeholders, it's time to get to the second concept: control structures. As you'll recall, they are always written between `<%` and `%>`.

Handle with care

If you work with control structures, note that they can easily break, for instance if you forget a space or closing tag. This can also happen with placeholders, but a lot more easily with control structures:



```
Parse error: syntax error, unexpected '}' in C:\Webserver\silverstripe2\silverstripe-cache\.cache.themes.bar.templates.Page.ss on line 23
```

It will happen but don't worry; we'll take a look at how to easily debug any template related errors. And the more experience you get, the less errors you'll make and you'll also find them much more quickly.

Whitespaces inside control structures matter: you must put a space after the `<%` and before the `%>`, but don't use tabs or newlines. Outside of the control structures, as between two tags that belong together, you can use any whitespace—just like with HTML.

Embedding and linking files

We've already covered the `<% include BreadCrumbs %>` control structure.

A very similar concept is the `<% require themedCSS(layout) %>`. It includes a CSS file, specifically from the theme's CSS directory. Again you should include the filename, without the file extension, as the only argument.

The advantages over a static `<link rel="stylesheet" type="text/css" href="/themes/blackcandy/css/layout.css">` are that you don't have to specify the path. Additionally, SilverStripe ensures that the file is referenced only once, even if you include it multiple times with `require`.

Additionally, one other neat trick is used to save you from the problem of CSS files being cached by the browser. A URL parameter is automatically attached by SilverStripe to the file. It stays the same as long as the file is unchanged and changes as soon as the file is edited. URL parameters for plain CSS files don't make any sense, but the browser doesn't know that. It simply sees that a different file is referenced and reloads it, regardless of any cache settings. So the actual output of a CSS file might look like this:

```
<link rel="stylesheet" type="text/css" href="http://localhost/themes/
blackcandy/css/layout.css?m=1271032378" />
```

If you want to include a CSS file from outside the theme's CSS directory, you can use `<% require css(sapphire/thirdparty/tabstrip/tabstrip.css) %>`. This is useful if the file is already included in one of the core directories so you don't need to keep multiple copies. Be sure not to use a leading slash, as the file won't be found this way. Start with the name of the base directory.

> But don't start replacing all your CSS references just yet. We'll cover another technique for that in the next chapter, which provides some additional features.

For including a JavaScript file, instead of the plain HTML way, you can use `<% require javascript(sapphire/thirdparty/jquery/jquery-packed.js) %>`.

Comments and base tag

If you want to add a comment to the template, use `<%-- My first comment... --%>`. The advantage over a regular HTML comment is that it is not included in the final output.

Instead of setting the base tag via `<base href="$BaseHref"></base>` you can simply use `<% base_tag %>`. This will also take care of browser specific requirements, providing the following output:

```
<base href="http://localhost/"><!--[if lte IE 6]></base><![endif]-->
```

Conditional statements

The first control structures we saw in the base `Page.ss` were conditional statements. The basic structure is:

Downloading the example code for this book

The author mantains an updated repository for this title's code bundle, downloadable from `https://github.com/xeraa/silverstripe-book`. You can also visit `http://www.PacktPub.com/support` for additional support.

```
<% if ConditionA %>
  ConditionA has been met.
<% else_if ConditionB %>
  ConditionA hasn't been fulfilled but ConditionB has.
<% else %>
  Neither ConditionA nor ConditionB were true.
<% end_if %>
```

If the placeholder `ConditionA` exists and is neither empty nor false, the first text will be inserted into the final HTML page. If the first condition doesn't fulfill these requirements, but the second one does, the second text is used. If neither conditions can be met, the third text is presented.

Note that the `else_if` and `else` parts are optional. As a minimum you only need the opening `if` and the closing `end_if`. If the control structure consists of two or more tags it's always closed off with an `end_` followed by the control's opening element.

So what can be a condition? For the moment let's just focus on placeholders. Take a look at the base `Page.ss`, what does this accomplish?

```
<% if MetaTitle %>
  $MetaTitle
<% else %>
  $Title
<% end_if %>
```

It means that, if a meta-title has been set: use it; otherwise take the regular title. Note that `MetaTitle` in the `if` statement doesn't start with a $. You mustn't use a $ in a control structure, only outside of it.

Another practical example for conditional statements can be:

```
<% if CurrentMember %>
  <h3>Welcome Back, $CurrentMember.FirstName</h3>
<% end_if %>
```

You can use the comparison operators == and != as well as the Boolean && and || in conditional statements. Other than that the template engine is rather limited. You can't use negations, comparison operators such as < or >, brackets to group conditions, or more than one Boolean operator.

That's both a boon and a bane: you are limited to certain statements (at least in the template), but your templates will always be short and clean. You cannot cram too much logic into them. We'll use Controller and Model for that.

To add a message for people not logged in, create an empty `if` block and put your message into the `else`. To avoid unnecessary whitespace put `if` and `else` in the same line:

```
<% if CurrentMember %><% else %>
  Hello Anonymous! Please log in to continue...
<% end_if %>
```

Control statements

Using control statements, you can access multiple records from a data set. The basic structure is:

```
<% control element %>
  Content of your control statement.
<% end_control %>
```

You can use a control statement on a single record, like `$CurrentMember` and use its elements like directly accessible placeholders:

```
<% control CurrentMember %>
  Hello $FirstName $Surname!
<% end_control %>
```

Note the similarity to placeholders in general in this case:

```
Hello $CurrentMember.FirstName $CurrentMember.Surname!
```

Both will output your name, for example, **Hello Philipp Krenn!**

What else can we do with control statements? Quite a lot, especially when working with navigation elements. The following code fragments are all intended to be used within the opening `control` block.

`AllChildren`	Get all subpages of the current page, even if they are not shown in the menu.
`Breadcrumbs`	Similar to the `$Breadcrumbs` placeholder but the control gives you more power while the functionality is the same.
`Breadcrumbs(  )`	`Breadcrumbs(2)` will show a maximum of two pages in the breadcrumb, useful if the available space is limited.
`Breadcrumbs(  ,  )`	`Breadcrumbs(3, true)` shows a maximum of three pages and all of them are just text—no pages are linked.
`Children`	Get all subpages of the current page, excluding elements hidden in the menu (use `AllChildren` to get those).
`ChildrenOf(  )`	Fetches all subpages of the provided page URL. For example, `ChildrenOf(home)` returns all subpages of the root page. Note that the argument is not quoted.
`Level(  )`	Use it to get access to a specific page in your current path. For example, if you are on the page `/home/news/2010`, `Level(2)` will give you access to the `news` page. Note that you can also use the placeholder `$Level(2).Title` to get the title, and so on.
`Menu(  )`	Use for getting all menu items of a specified level. `Menu(1)` provides access to the main menu entries.
`Page(  )`	Gets access to the specified page by providing its URL. For example: `Page(home)`.
`Parent`	Get the parent page. You can also use `$Parent.Title`, and so on.

Furthermore, there are some very handy elements for styling our records. Use these inside the control block:

- Zebra-striping: `$EvenOdd` returns `even` or `odd`, depending on the row number. Alternatively, you can use `<% if Odd %>` and `<% if Even %>`.
- You can achieve the same and a little more with `$MultipleOf(Number, Offset)`, which returns a Boolean value. For example, to insert a dot after every fourth element and for all others a comma call: `<% if MultipleOf(4) %>.<% else %>,<% end_if %>`

- In order to target every second entry within the groups of four from the statement above, use `$Modulus(Number, Offset)`, which returns a number. By calling `$Modulus(4)` you'll get `1`, `2`, `3`, `0`, `1`, `2`,... , which can be easily used for styling within an HTML class attribute. Take special note of the initial value of 1 as most IT systems start with 0. However, this is more intuitive for most designers, who will generally focus on the templates.
- Counting can be done with `$Pos`, starting at 1 again, while `$TotalItems` returns the number of rows in the set.
- First, last, and middle elements: `$FirstLast` will be `first`, `last`, or empty. Alternatively, you can use `<% if First %>`, `<% if Middle %>` and `<% if Last %>`.
- If you are inside a control statement, you can only access locally available elements. If you need an outside value, prefix it with `$Top`. For example, `$Top.URLSegment` will give you the page's URL segment. `$URLSegment` alone might return nothing, assuming you're inside a control block.

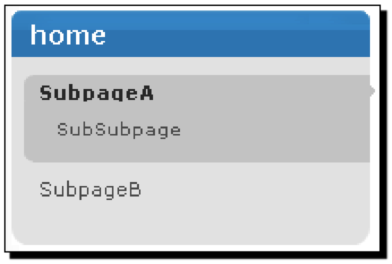
BlackCandy revisited

Let's get back to one of the more complex parts of the BlackCandy theme, which we've left out so far. By now you should easily understand what's going on, so open up the file `SideBar.ss` in the folder `themes/blackcandy/templates/Includes/` (the relevant part is below):

```
<ul id="Menu2">
  <% control Menu(2) %>
    <% if Children %>
      <li class="$LinkingMode">
      <a href="$Link" title="Go to the $Title.XML page"
        class="$LinkingMode levela">
        <span><em>$MenuTitle.XML</em></span>
      </a>
    <% else %>
      <li><a href="$Link" title="Go to the $Title.XML page"
            class="$LinkingMode levela">
      <span><em>$MenuTitle.XML</em></span></a>
      <% end_if %>
      <% if LinkOrSection == section %>
        <% if Children %>
          <ul class="sub">
            <li>
              <ul class="roundWhite">
                <% control Children %>
                  <li><a href="$Link"
                  title="Go to the $Title.XML page"
```

```
                     class="$LinkingMode levelb">
                     <span><em>$MenuTitle.XML</em></span></a></li>
                   <% end_control %>
                 </ul>
               </li>
             </ul>
           <% end_if %>
         <% end_if %>
       </li>
     <% end_control %>
   </ul>
```

Creating two pages (**SubpageA** and **SubpageB**) on the second level and one on the third (**SubSubpage**, child of **SubpageA**), the output looks like this:



The page source will read as follows:

```
<ul id="Menu2">
  <li class="current"><a class="current levela"
    title="Go to the SubpageA page" href="/home/subpagea/">
    <span><em>SubpageA</em></span></a>
    <ul class="sub">
      <li>
        <ul class="roundWhite">
          <li><a class="link levelb"
              title="Go to the SubSubpage page"
              href="/home/subpagea/subsubpage/">
              <span><em>SubSubpage</em></span></a></li>
        </ul>
      </li>
    </ul>
  </li>
  <li><a class="link levela" title="Go to the SubpageB page"
       href="/home/subpageb/"><span><em>SubpageB</em></span></a>
  </li>
</ul>
```

Here's an explanation of the code:

- First, we're getting the second-level menu items: `<% control Menu(2) %>`.
- Then we're checking if the current item has sub-items, which is true for **SubpageA** but not **SubpageB**, adding some additional styles:

  ```
  <% if Children %>
  ```
- If you currently are on a page with children or on one of the child pages themselves, expand them (this is the case in the image above):

  ```
  <% if LinkOrSection == section %>
  ```
- Finally, loop over all child elements and output them:

  ```
  <% control Children %>
  ```

Creating our own theme

Now that we've covered all the basics, let's create and then edit the necessary folders and files for a theme of our own:

Time for action – files and folders for a new theme

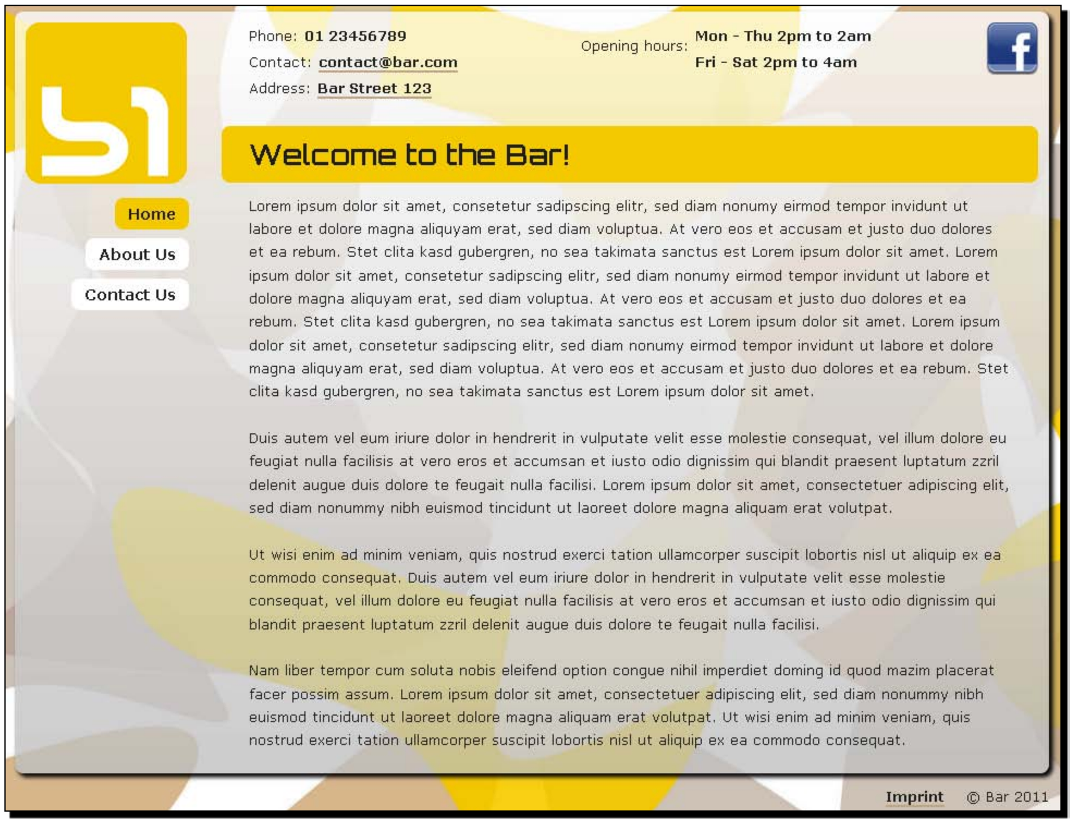
We could start off from scratch, but let's not make it more complicated than necessary:

1. Simply copy the BlackCandy theme to a new folder in the same directory. We'll just call it `bar` as the page should be for a bar business.
2. Now reload the CMS backend and you should be able to switch to the new bar theme.
3. As we won't use any of the available images we can delete all image files, but leave the folder intact—we'll add our own pictures later on.

Basic layout

Next, we'll create our page's general layout. We'll further improve and extend it in the next chapters, but we already have the foundation of the example project used throughout the book. Time to put our template skills into action!

Here's the basic layout we want to create:



Phone: 01 23456789
Contact: contact@bar.com
Address: Bar Street 123

Opening hours: Mon - Thu 2pm to 2am
Fri - Sat 2pm to 4am

Welcome to the Bar!

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

© Bar 2011

So far it's a pretty basic page. We'll take a good look at the templates and basic structure but won't spend much time on the HTML and CSS specifics. Do, however pay attention on how to set up the different CSS files to make the most out of them.

We won't list every line of code involved. If you'd prefer, you can simply copy the missing parts from the code provided for the book at: `http://www.packtpub.com/support`

File themes/bar/templates/Page.ss

This page is pretty empty. We'll later add an intro page with a different structure, so the code these templates will share is rather limited.

Time for action – the base page

Add the following code to the file `themes/bar/templates/Page.ss`:

```
<!doctype html>
<html lang="$ContentLocale">
<head>
  <meta charset="utf-8"/>
  <% base_tag %>
  <title>
    <% if MetaTitle %>
      $MetaTitle
    <% else %>
      $Title
    <% end_if %>
  </title>
  $MetaTags(false)
  <link rel="shortcut icon" href="favicon.ico"/>
  <!--[if lt IE 9]>
    <script src="http://html5shim.googlecode.com/svn/trunk/html5.js">
    </script>
  <![endif]-->
</head>

<body>
  $Layout
  <noscript>
  <br/>&nbsp;<br/>&nbsp;<br/>&nbsp;<br/>&nbsp;<br/>&nbsp;<br/>&nbsp;
    <div><p>
      <b>Please activate JavaScript.</b><br/>
      Otherwise you won't be able to use all available functions
       properly...
    </p></div>
  </noscript>
</body>
</html>
```

What just happened?

The highlighted parts are SilverStripe specific:

- First, we set the language of our content. Depending on what you've selected during the installation process this can vary. By default, it will be: `<html lang="en-US">`.

- Setting the page's base; this can be useful when referencing images and external files. Depending on how you set up your system this will vary.
- Setting the title just like in the BlackCandy theme.
- Next, adding `meta` tags without the title element. These are only set if you actually enter them in the CMS on the **Metadata** tab on the desired page's **Content** tab. Empty fields are simply left out in the output. Only the SilverStripe note is set in any case.
- Finally, including the page's specific layout.

The JavaScript we include for Internet Explorer versions before 9 fixes their inability to correctly render HTML5 tags. To learn more about the specific code, head over to `https://code.google.com/p/html5shim/`.

Why HTML5?

The motivation behind it isn't to stay buzz-word compliant. While HTML5 is not a finished standard yet, it already adds new features and makes the development a little easier. Take a look at the `doctype` declaration for example: that's much shorter than XHTML. Now there's a real chance of actually remembering it and not needing to copy it every time you start from scratch. New features include some very handy tags, but we'll come to those in the next code sample.

The output on the final HTML page for the second and third elements (we've already taken a look at the first) in the header looks like the following listing. Which part in the template is responsible for which line or lines?

```
<title>home</title>
<meta name="generator"
      content="SilverStripe - http://silverstripe.org" />
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<meta name="keywords" content="some keyword" />
<meta name="description" content="the description of this page" />
```

Did you notice that there is no reference to a CSS file and the styling is still applied? Remember that CSS files are automagically included if you stick to the naming convention we've already discussed.

File themes/bar/templates/Layout/Page.ss

Now let's move beyond the basics and do something a bit more interesting.

Time for action – the layout page

Add the following code to the file `themes/bar/templates/Layout/Page.ss`:

```
<div>
  <img src="$ThemeDir/images/background.jpg" alt="Background"
id="background"/>
</div>
<section id="content" class="transparent rounded shadow">
  <aside id="top">
    <% include BasicInfo %>
  </aside>
  <% include Menu %>
  <section class="typography">
    $Content
    $Form
  </section>
</section>
<% include Footer %>
```

What just happened?

We rely on just three includes which are very easy to reuse on different pages.

We then reference the page's content and possible forms (no comments will be required).

The only template part we've not yet covered is the `$ThemeDir`. This is replaced by the path to our theme, in our case `themes/bar/`. So you don't need to worry about hard-coded paths when copying template files between different themes. You only need to take care of paths in the CSS files as they are not processed by the template engine.

The includes: BasicInfo.ss, Menu.ss, and Footer.ss

In the previous code segment, we've referenced three includes. Let's not forget to create them.

Time for action – the includes

The includes we're yet to explore are `Menu.ss`, `BasicInfo.ss`, and `Footer.ss`.

1. The menu is very simple, we only include the first level, highlighting the currently active item, as we won't have more subpages:

   ```
   <nav id="menu">
     <ul>
       <% control Menu(1) %>
   ```

```
      <li class="$Linkingmode">
        <a href="$Link">$MenuTitle</a>
      </li>
    <% end_control %>
  </ul>
</nav>
```

2. `BasicInfo.ss` only contains the `$ThemeDir` placeholder besides good old HTML:

```
<a href="home">
  <img src="$ThemeDir/images/logo.png" alt="Logo" id="logo"/>
</a>
<ul id="details-first">
  <li>Phone: <b>01 23456789</b></li>
  <li>Contact: <a href="contact">contact@bar.com</a></li>
  <li>Address: <a href="location">Bar Street 123</a></li>
</ul>
<div id="details-second">
  <div class="left">Opening hours:</div>
  <div class="right"><p>
    <b>Mon - Thu 2pm to 2am</b><br/>
    <b>Fri - Sat 2pm to 4am</b>
  </p></div>
</div>
<a href="http://www.facebook.com/pages/">
  <img src="$ThemeDir/images/facebook.png" alt="Facebook"
    id="facebook"/>
</a>
```

Be sure to replace `bar.com` with your own domain. However, using it here is quite fitting because we're making a page for a bar and `http://www.bar.com` is only a placeholder itself. It's derived from foobar or foo bar, a placeholder name often used in computer programming. But take a look at the site yourself for more information.

3. Finally, the footer makes use of the `$Now` placeholder and the date formatter `.Year` to retrieve the current year:

```
<footer>
  <span><a href="imprint">Imprint</a></span>
  <span>&copy; Bar $Now.Year</span>
</footer>
```

Have a go hero – create the layout and pages

Now that we've gone over all of the content, it's time to build the page. Feel free to copy some code as we're not a typing class, but take careful note of every SilverStripe specific part, so you understand what you're doing.

We use three includes, so create these files and remove the rest still available from BlackCandy as well as the unused `templates/Layout/Page_results.ss`.

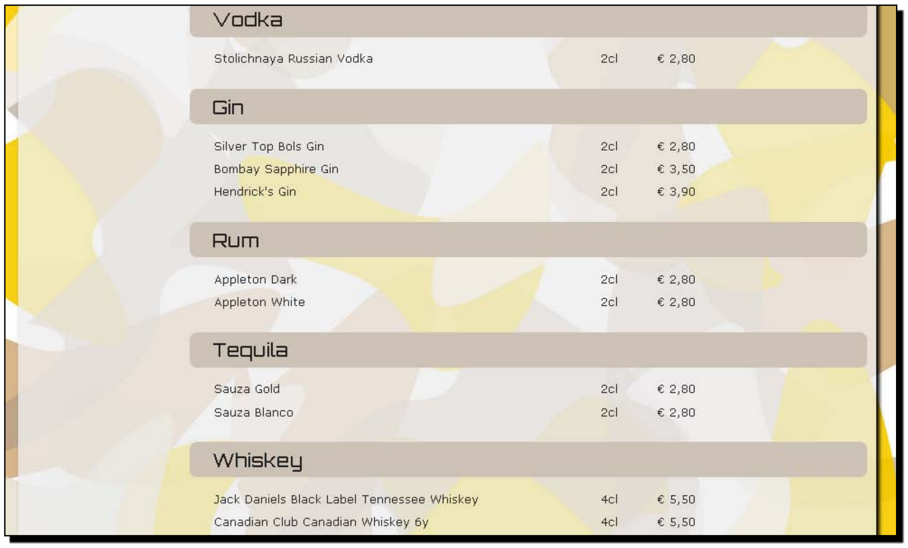
And don't forget to flush the cache by appending `?flush=all` to the page's URL! When working with includes it will save you a lot of trouble.

We've hardcoded some links to different pages in the template files. Make sure you add all necessary pages (including the imprint, but don't add it to the menu), we'll use them in the next chapters!

What's next?

Now that we've created our general content page, we can already do quite a lot using the CMS:

- Write an interesting **Home** and **About Us** page
- Provide all the necessary information for the **Imprint** and how to **Contact Us**
- Advertise the program and daily specials of your bar
- List all your drinks and cocktails in a table



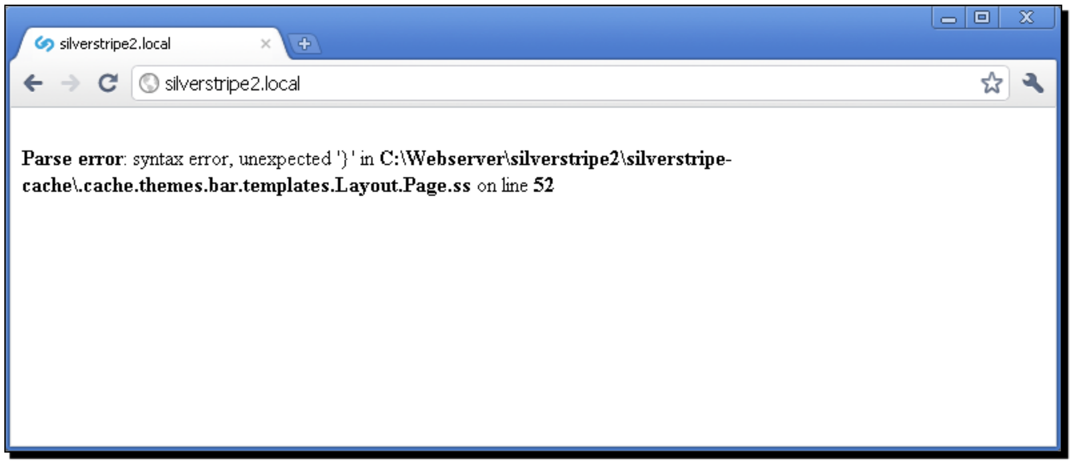
Vodka		
Stolichnaya Russian Vodka	2cl	€ 2,80
Gin		
Silver Top Bols Gin	2cl	€ 2,80
Bombay Sapphire Gin	2cl	€ 3,50
Hendrick's Gin	2cl	€ 3,90
Rum		
Appleton Dark	2cl	€ 2,80
Appleton White	2cl	€ 2,80
Tequila		
Sauza Gold	2cl	€ 2,80
Sauza Blanco	2cl	€ 2,80
Whiskey		
Jack Daniels Black Label Tennessee Whiskey	4cl	€ 5,50
Canadian Club Canadian Whiskey 6y	4cl	€ 5,50

Let your creativity run free and start working with our project!

Not finding the #*?~^ error?

When you are creating your own theme, you're bound to make mistakes so don't panic.

For example, if you write `<%control Menu(1) %>` in your template (note the missing space before `control`), your error message will look something like this:

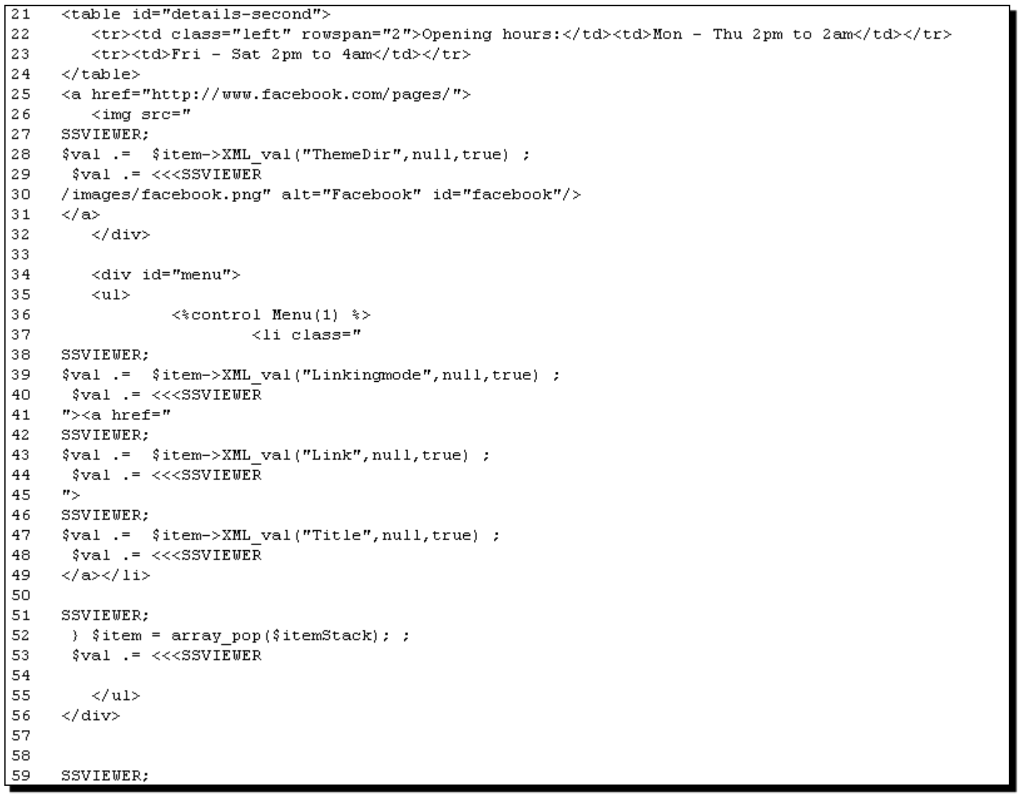


```
Parse error: syntax error, unexpected '}' in C:\Webserver\silverstripe2\silverstripe-
cache\.cache.themes.bar.templates.Layout.Page.ss on line 52
```

While it clearly points to the correct file, the } and line number are puzzling as `templates/Layout/Page.ss` does not contain any curly braces nor is it anything like 52 lines long.

Remember that the template engine replaces the control code with plain PHP. So if there is an error in the template, the resulting PHP code is not correct. The web server, trying to interpret it, spits out this error message relating to the PHP code.

In order to see the PHP code generated from the template, attach `?isDev=1&showtemplate=1` to the current URL. If you're not already logged in, you'll be prompted to do so. This is a quick helper like flushing the cache. It forces the page into development mode and shows the generated template code:



```
21   <table id="details-second">
22      <tr><td class="left" rowspan="2">Opening hours:</td><td>Mon - Thu 2pm to 2am</td></tr>
23      <tr><td>Fri - Sat 2pm to 4am</td></tr>
24   </table>
25   <a href="http://www.facebook.com/pages/">
26      <img src="
27   SSVIEWER;
28   $val .=  $item->XML_val("ThemeDir",null,true) ;
29    $val .= <<<SSVIEWER
30   /images/facebook.png" alt="Facebook" id="facebook"/>
31   </a>
32      </div>
33
34      <div id="menu">
35      <ul>
36              <%control Menu(1) %>
37                      <li class="
38   SSVIEWER;
39   $val .=  $item->XML_val("Linkingmode",null,true) ;
40    $val .= <<<SSVIEWER
41   "><a href="
42   SSVIEWER;
43   $val .=  $item->XML_val("Link",null,true) ;
44    $val .= <<<SSVIEWER
45   ">
46   SSVIEWER;
47   $val .=  $item->XML_val("Title",null,true) ;
48    $val .= <<<SSVIEWER
49   </a></li>
50
51   SSVIEWER;
52    } $item = array_pop($itemStack); ;
53    $val .= <<<SSVIEWER
54
55      </ul>
56   </div>
57
58
59   SSVIEWER;
```

By taking a quick look at the result you'll see multiple `SSVIEWER` replacements representing the interwoven PHP statements and in our specific case:

```
35  <ul>
36    <%control Menu(1) %>
37      <li class="
```

So there's a control structure which hasn't been replaced. There must be something wrong here!

The actual error is caused by the closing } on line 52 (being the replacement for the `<% end_control %>`) as the opening { doesn't exist. This should make it easy to figure out the problem.

So for debugging `?isDev=1&showtemplate=1` is your best friend, use it before poking in the dark.

Adding interactive features

While we've created the basic layout, the site is a bit static. To make it a little more interactive, we'll allow visitors to easily share a link to our page on Facebook and Twitter. There are JavaScript libraries for easy integration, but without scripting enabled they don't work at all. Additionally, we don't want to rely too much on external code. So for full control we had better roll our own sharing service.

Twitter and Facebook both follow a rather simple schema for sharing content: link to a specific page of the social network, provide the current URL as a parameter, and that's it. Once a user clicks on such a link, they must log into their social network's account and can then share the URL, optionally adding more text at their liberty.

Time for action – let users share our page on Facebook and Twitter

Let's integrate these links into our templates. We'll use another include for optimal flexibility:

1. Create a new include template called `Share.ss`. (Don't forget that includes are located in `themes/bar/templates/Includes/`):

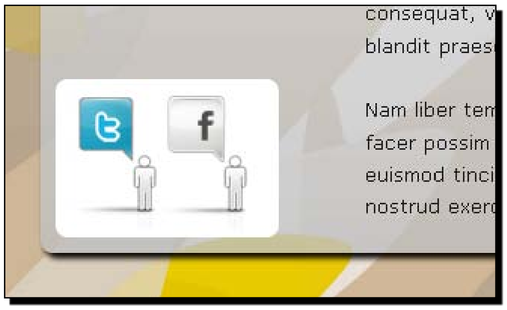
```
<aside id="share" class="rounded">
  <a href="http://twitter.com/home?status={$AbsoluteLink}"
    class="twitter" target="_blank">Twitter</a>
  <a href="http://www.facebook.com/sharer.php?u={$AbsoluteLink}"
    class="facebook" target="_blank">Facebook</a>
</aside>
```

2. Reference it in the `themes/bar/templates/Layout/Page.ss` file just below the menu. That's also where we want to place it in the final layout, on the left-hand side in the content area's bottom left corner:

```
<div><img src="$ThemeDir/images/background.jpg" alt="Background"
id="background"/></div>
<section id="content" class="transparent rounded shadow">
  <aside id="top">
    <% include BasicInfo %>
  </aside>
  <% include Menu %>
  <% include Share %>
  <section class="typography">
    $Content
```

```
      $Form
    </section>
  </section>
  <% include Footer %>
```

3. Flush the page (`?flush=all` but you know that already) and you should now see a new element on your page. For the screenshot, some CSS has been added along with images to pep it up a little, but regular text links will accomplish just the same.



The only remaining question is, in which file should you add the CSS and into which folder should you save the images? We'll cover that right in the next section.

What just happened?

Without going into too much detail on the Twitter and Facebook APIs: `http://twitter.com/home` is the address for sharing content on Twitter. Using the status parameter you can add some text to the message, in our case the URL. But we need the complete address, including the `http://` for the services to know what we mean. And that's exactly what `$AbsoluteLink` provides. So the raw output of our include could be:

```
<aside id="share" class="rounded">
  <a href="http://twitter.com/home?status=http://localhost/contact-
us/" class="twitter" target="_blank">&nbsp;</a>
  <a href="http://www.facebook.com/sharer.php?u=http://localhost/
contact-us/" class="facebook" target="_blank">&nbsp;</a>
</aside>
```

The curly braces in the template are not strictly needed in this case and are automatically removed as they delimit a placeholder. They only help the template engine to determine where the placeholder really starts and ends. In order to avoid confusion, you should always use them when a placeholder isn't delimited by spaces.

Assume you have a placeholder `$Filename` that gets you a filename, but as the extension is always PNG you simply add that in the template, leaving you with `$Filename.png`. In this case the curly braces will be mandatory as the template engine would think you have an object `Filename` and want to access its attribute `png`. `{$Filename}.png` will be the solution in this case. Don't save on curly braces, they do no harm and improve readability for yourself and SilverStripe.

Coming back to our earlier questions:

- `layout.css` is the right CSS file to use for this styling as it's used in the general layout and is not needed in the CMS
- Images should be added to `themes/bar/images/` or possibly a subfolder

Taking care of search engines

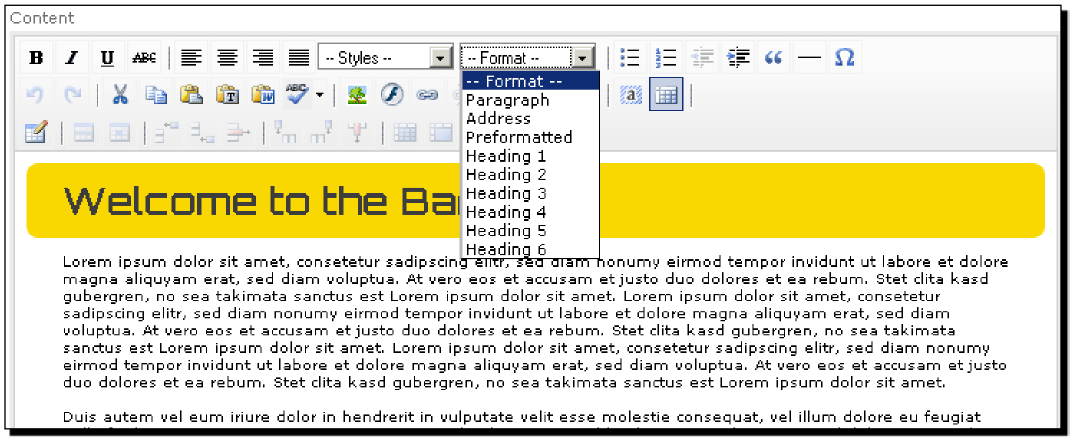
Now that we can create dazzling layouts, it's time to consider how users can find our page. We'll only cover aspects concerning SilverStripe—if you're interested in every detail of the topic, get a book dedicated to **search engine optimization** (**SEO**).

The most important rule for getting good results in search results is that content is king. Focusing on the substance of your page gives you a strong base to build upon. After that, you can further improve it.

Semantic HTML

The structure of your content is important. Use the right tags to give the correct meaning (semantics) to your information. This enables machines, specifically search engines, to easily understand it. For example, use `<h1>` for the main heading of your page, `<h2>` for subheadings, and so on. Don't leave a level out or build your own formatting rules with `<div class="h1">`. SilverStripe's WYSIWYG editor makes it easy to use the right markup.

> How does the CMS know how to style our headings? By adding that information to `typography.css` we've automagically added it to the CMS as well. If you put all your styling information into `layout.css`, you won't be able to benefit from that convention.



Welcome to the Ba

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat

Meaningful URLs

Use human-readable URLs, for example, `/contact` instead of `/index.php?pageid=12`. On the one hand search engines value the word "contact", so if someone searches for a contact page and your brand, you have a better chance of being found. On the other hand you might be penalized for using URL parameters as you can provide the same content only with different advertising or layout.

If your web server's configuration permits it, SilverStripe will always default to using human-readable URLs, using the page's title to generate a unique URL, but you can of course change it.

Broken links

Neither search engines nor users will appreciate it if you have broken links on your page.

If internal links within a WYSIWYG area are broken, SilverStripe will mark them with a red background. Additionally, you can use the broken links report of your installation under the URL `/admin/reports/`, to see if there are any.

If you link to external pages or use hardcoded links in your template (as we did before), you'll need to use an external tool such as `http://validator.w3.org/checklink/`.

When you're using the built-in page selector for setting up internal links, you're not linking to a specific URL but actually to a page ID. So even if you manually change the URL of a page, links are automatically updated. You don't need to change them by hand.

Duplicate content

If you want to use the same page in multiple places of your site, use the page type **Redirect Page** or **Virtual Page** for actually mirroring it.

A redirect page returns a HTTP status code "301 Moved permanently" and should be the preferred method. This also frees you from making changes in multiple places.

Meta tags

Search engines won't pay too much attention to meta tags as they have been widely abused in the past. The most important one is the description as most search engines use it as the teaser text in the result list. If no description is found some text is grabbed from the page and used instead, producing varying results. If you can't remember how to enter the information in the CMS or how to place it in the template, see earlier in this chapter.

Validation

Your website should follow the specifications for the HTML5 and CSS3. This will help both the browsers to display your content correctly, and search engines to interpret it the way you intended. For example, if you don't close a heading, it must be guessed what should be included and what not. Don't leave a machine guessing at the right interpretation of your messy code.

SilverStripe enables you to write validating code. Regularly check it with a validator—for example, using `http://validator.w3.org/` or a browser plug-in.

Sitemap

A sitemap is an XML file, telling search engines which pages exist on your site, when they were last changed, and how important they are. This was first introduced by Google but more or less every other search engine has adopted the same system.

We've already discovered the Google sitemap module while taking a look at the file structure; it's included and activated by default. In order to view the XML data generated by it go to the page `/sitemap.xml` in your installation. Just like any other page in your system there is no concrete file for it, it's generated by the system on the fly. We'll discuss the details of that later.

To exclude a page from the index (a search engine will then completely ignore the page) or change the importance of a page, go to the **Google Sitemap** tab on the desired page's **Content** tab.

Pop quiz – say hello to the current user

Take a look at the following template snippet:

```
<% if $CurrentMember %>
  Hello $CurrentMember.FirstName $CurrentMember.Surname!
<% end_if %>
```

Assume you are currently logged into the CMS and have set both the first and surname for the current user. What will this output?

1. The placeholder `$CurrentMember` doesn't exist, so this won't output anything.
2. It will output your name, for example **Hello Philipp Krenn**.
3. This will cause an error because of the first line.

Summary

We've learned a lot in this chapter about templates:

- How to use them and the engine which powers them.
- What themes are, where you can download existing ones and how you can activate them on your page.
- SilverStripe's template engine, how it's designed and what its main features are.
- How to style the content of your page.
- BlackCandy: the default theme, its major parts, and how they function.
- The template's controls, how they work and how to use them.

- Building our own theme which we'll extend and hone over the next few chapters—showing how easy that actually is with SilverStripe.
- Debugging errors in templates—if you know how to do it, it's not really that hard.
- How to enable users to interact with Facebook and Twitter on our page.
- Finally, how to optimize your page for search engines.

We've also discussed some limitations and specific features of SilverStripe's template system.

Now that we've learned about templates, we're ready to add our own placeholders, define custom page types and add some logic. This is the topic of the next chapter.

3
Getting "Control" of Your System

Having created a nifty layout is a good start, but now it's time to get a better understanding of the system and add features. Right now we're barely scratching the surface of SilverStripe's capabilities, so hold tight and enjoy the learning process.

In this chapter we shall:

- Take a look at the architecture of SilverStripe
- Create our own pages
- Create some logic for new features of our pages
- Try out some URL tricks for debugging and more

So let's start with the underlying structure of the system, which we'll then extend by building an intro page, allowing visitors to share our page on Facebook or Twitter, and protecting our e-mail addresses against spam.

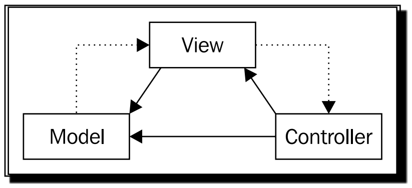
Model View Controller (MVC)

In the previous chapter we covered templates in depth, generally called the View. There are two more roles, namely Controller and Model. Together these three are generally abbreviated as **MVC**, which we'll explore over the next few pages. Don't be confused that we've started with the middle one—there is no strict order, it's just a naming convention.

Roles or layers?

Whether MVC consists of roles or layers is a matter of debate. We won't make a strict distinction and will use both terms interchangeably. Role is probably more appropriate, but layer is also widely used. However, note that information doesn't need to flow through each layer in every request, nor is it strictly passed from one level to the next in MVC.

The information flow between the roles is outlined by the following image; we'll be taking a look at each of them. Solid lines indicate a direct association whereas dotted lines indicate an indirect one, but we won't put too much emphasis on this difference:



Why the big fuss?

What are the reasons for neatly separating your application into three distinct roles?

- Having different roles makes it possible to split up tasks so that everyone can concentrate on their strengths. This allows teams of people to work collaboratively to build a project, working on different areas. If you're a designer, you only need to work with templates and don't need to mess around with the rest of the application. Also, having three separate roles minimizes possible side effects of less structured code as the information flow is constrained.
- Additionally, it makes it easy to reuse code, saving you time during development and especially during maintenance. Let's assume we want to hava a website for both humans to view it and for machines to automatically exchange information via XML. Thanks to MVC you only need to provide two different templates, while the same Controller and Model can be used for both. If you just had a single block of code, you'd need all of it twice.

MVC is a modern software technique, considered best practice and becoming very popular in modern web-applications. You'll find more of these patterns in SilverStripe, but this is the most central one.

So what tasks do these different roles perform and what do they not cover?

View: Presenting your content

As we saw in the previous chapter, the View renders information through a template into a form suitable for user interaction. In most cases this will be HTML or XHTML but it can also be XML, for processing by some other system. SilverStripe templates fully support XML output.

You can have multiple Views for a single Controller. For example, you can create a human-readable page in HTML and one in XML to be used by web-services. While the information made available will be the same in each case, the resulting output will be very different. This decouples the presentation from the underlying Model.

Model: Managing your data

The Model is used for managing the state of the application, that is, the data used in the system. This data can be temporary, only needed for a single request or during a session, or persistent. Persistent data is generally stored in a (relational) database and this is often the main use case, but the Model is not limited to it.

Besides saving, updating, deleting, and retrieving information, the Model also handles data validation along with relationships between multiple entities of your system. We'll focus on the Model in the next chapter.

Controller: Providing your logic

This chapter will primarily deal with the Controller. It handles the processing of information in the application and holds the system together. Inputs are passed from the View to the Controller, which will process the information (if needed) and then forward it to the Model. For a complex presentation of information, it might be necessary for the Controller to fetch data from the Model and then prepare it for the presentation in the View, but the two can also communicate directly.

Although the Controller provides the URL structure, authentication, information processing, and more, data validation is split up between Model and Controller. The Model should be used for general purpose validation, for example, that a given date doesn't contain text. Specific validation should be done in the Controller. For example, one Controller might require a specific period of time while another one doesn't, but both use the same field in the Model. How to split this up depends on your specific use case, and what you consider valid in general or only for a specific Controller.

Taking a look at an example

Consider a simple shopping cart. Users who are logged in can add items to their cart, within a certain limit, and the cart can return the sum of all items entered.

When calling the URL to enter a new item, the Controller will take your request and ensure you're logged in. It will then call the appropriate View, which in turn will load all the relevant information from the database. Once you've added an item and submitted your input to the Controller, it is forwarded to the Model. The Model then checks if the item has a correct price and is still within your limit. If that's the case, the information is saved to the database; otherwise, an error message is returned to the View.

If you want the current sum of all your items, go to the appropriate page by calling the right URL. The Controller will again process your request, fetching all items from the Model and calculating the sum total. Once this is done the result is forwarded to the View to display the final page.

Don't worry if you don't feel comfortable designing your own system right away. We'll take a look at more examples as we continue with our application.

Structure of a Page

Now that we've covered MVC in general, let's see how it's implemented in SilverStripe.

Page.php: Your initial start

Your custom code for Model and Controller is typically placed in the `mysite/code/` folder. The following example defines the basic structure for the file `mysite/code/Page.php`:

```
<?php
class Page extends SiteTree {
  // I'm the Model
}
class Page_Controller extends ContentController {
  // I'm the Controller
}
```

The closing ?> is missing!

Actually, it has been left out intentionally. You can safely do that in a PHP-only file without causing an error. Due to the MVC pattern we can be absolutely sure that there is nothing but PHP in this file. The big advantage of this approach is that you can't accidentally send content to the browser before the View has rendered, which can cause a number of errors. The View should be the only component sending content to the browser. So it's a good practice to leave the closing `?>` out and we'll stick to it.

All we have to do is defining two classes, one for the Model and one for the Controller. Some frameworks split the two into different files, but as they are closely related they're simply combined in SilverStripe.

In the next chapter, we'll see that not everything is a page and that you can define Models on their own, separate from a Controller. Nevertheless, if you create a page, the two layers are combined in one file - with two distinct classes.

As we've already said, we won't be covering object-orientation in detail, but this should help you get started: You can see that we've defined a class in our example. Let's just say that a **class** is a blueprint for creating concrete instances, called **objects**. So in the CMS the page class is used to create objects such as a contact page, a location page, and so on.

Convention over configuration revisited

Remember "convention over configuration", where sticking to certain rules provides added functionality? This is used a lot in SilverStripe to connect different files and functionality together.

The file is called `Page.php`, which implies:

- The Model class must be called the same (case sensitive), excluding the file extension: `class Page`
- The Controller class is also called the same but with `_Controller` attached: `class Page_Controller`
- The template for this page is stored in a file of the same name: `Page.ss`, preferably as a reusable theme in the `themes/` directory

So if you wanted to create a test page, you would create a file `mysite/code/TestPage.php`. In it you would create two classes. One for the Model `class TestPage` and one for the Controller `class TestPage_Controller`. Finally, you would create a file to provide a page-specific layout `themes/bar/templates/Layout/TestPage.ss` (assuming you wanted to extend your custom theme).

Therefore creating your own pages, using the MVC pattern, is actually pretty simple. Just one more thing to point out: "Page" is special. It's a convention similar to the start page of your application that has the URL `/home`—stick to it because:

- `mysite/code/Page.php` is generally used for extending all other pages, so if you want something to be available or happen everywhere in your system, you can put it here. You'll see an example in a few moments.
- `themes/bar/templates/Page.ss` is the basic template into which every other page's layout is inserted, as long as it isn't specifically overridden. See the previous chapter if you can't remember how templates are combined.
- `themes/bar/templates/Layout/Page.ss` is the default template. If you create your Controller and Model file, but don't set up a concrete View file, `Page.ss` will be used as a fallback.

Extending classes is key

So far we've left out the "extend" part of the class definitions. Let's look at this now.

`Page` inherits from `SiteTree`, which is a SilverStripe core class, providing the Model for our application. Every subclass of `SiteTree` must also define a Controller and we use the `ContentController` for that.

Any other class will generally extend these two classes. If you need to, you can have a class inherit from one of the previously extended pages.

Going back to our test page, we'd have `class TestPage extends Page` and for the Controller `class TestPage_Controller extends Page_Controller`.

Coding conventions

Strictly speaking, coding conventions are not necessary, but they can make the development and particularly the maintenance of projects easier. As the SilverStripe core follows a set of coding conventions it is advisable to follow them as well, so we'll gradually introduce them as we go along. In your own projects it's advisable to follow them; if you provide patches or enhancements to the core, you're required to do so.

- Write class and file names in UpperCamelCase, for example, `TestPage.php` and `class TestPage`.
- Try to use unique class names. If you don't, be prepared to see error messages.
- A class must reside in a file of the same name, not counting anything following an underscore. So within `mysite/code/TestPage.php` you can have `class TestPage`, `class TestPage_Controller`, `class TestPage_Helper`, and so on.
- Start off PHP files with `<?php` and not the short-hand form `<?`. And remember, leave out the closing `?>` in a PHP script.

The naming convention is also used for **autoloading** files. Generally `require()` and `include()` are no longer necessary. The needed classes and templates are automagically found and used if filename and class name match.

Have a go hero – creating our own page

After covering all the theory it's time to put our new knowledge into action. We'll keep using the same example system as in the previous chapter so that we already have a layout and some pages.

First open up the file `mysite/_config.php` and add the following line of code to it:

```
Director::set_environment_type('dev');
```

By default, your page is in live mode. In case you're actually working on the code like we are now, this isn't ideal as error messages are not shown. To change into development mode, you need to add the above line to the configuration file. We'll come back to the configuration later, but for the moment this will do.

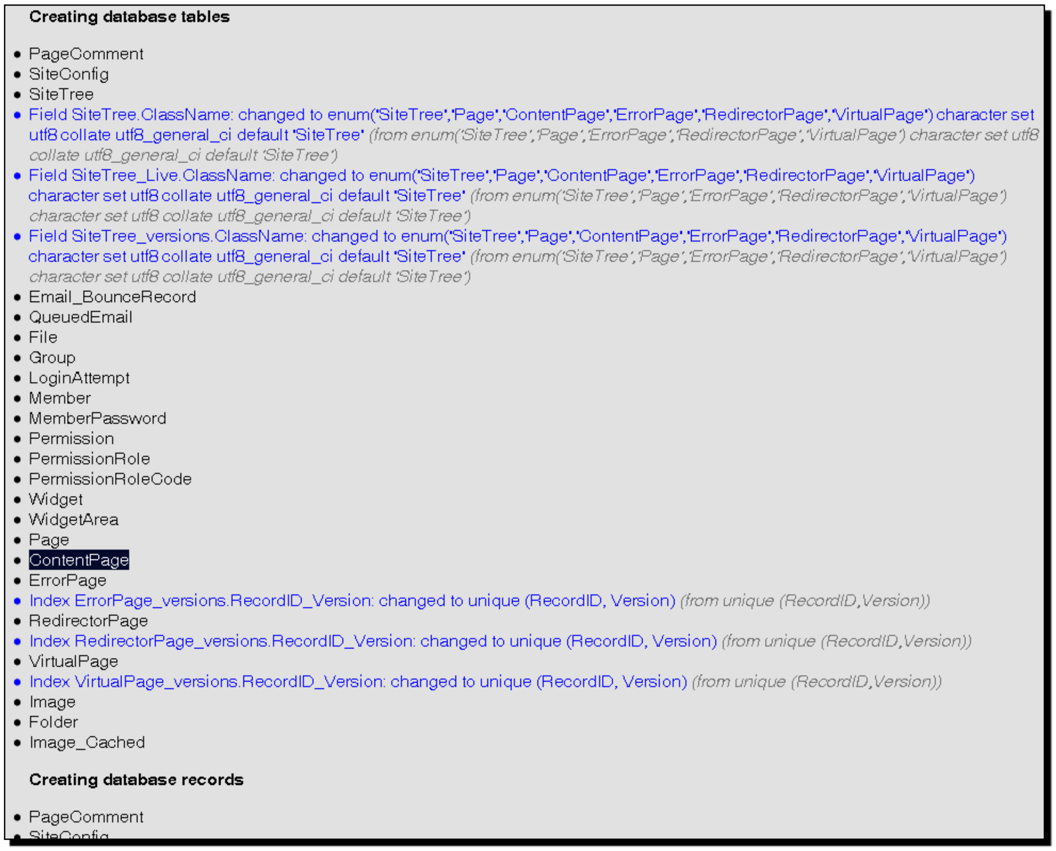
Next, create a new file `ContentPage.php` in the `mysite/code/` folder. Add both Controller and Model classes to it, which inherit from `Page`. This page will be used for all general content pages. Right now it doesn't add any functionality, so the code should be:

```
class ContentPage extends Page { }
class ContentPage_Controller extends Page_Controller { }
```

You don't need to create a template for the new page, as falling back to the default one is fine for now: in case there isn't a Controller specific template (`/themes/bar/templates/Layout/ContentPage.ss` in our case) then `themes/bar/templates/Layout/Page.ss` will be used.

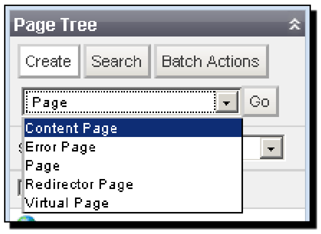
While the `themes/` folder is left unchanged, `mysite/code/` should now contain the files `Page.php` and `ContentPage.php`.

After saving the PHP file, go to the URL `/dev/build`. So if your installation is located at `http://localhost` it will be `http://localhost/dev/build`. Remember that we said SilverStripe synchronizes your code and database? By going to this URL you simply tell the framework to do just that. The system will look for any changes and apply them. So in our case the output should look like the following image. Note the highlighted **ContentPage**:



```
Creating database tables

• PageComment
• SiteConfig
• SiteTree
• Field SiteTree.ClassName: changed to enum('SiteTree','Page','ContentPage','ErrorPage','RedirectorPage','VirtualPage') character set
  utf8 collate utf8_general_ci default 'SiteTree' (from enum('SiteTree','Page','ErrorPage','RedirectorPage','VirtualPage') character set utf8
  collate utf8_general_ci default 'SiteTree')
• Field SiteTree_Live.ClassName: changed to enum('SiteTree','Page','ContentPage','ErrorPage','RedirectorPage','VirtualPage')
  character set utf8 collate utf8_general_ci default 'SiteTree' (from enum('SiteTree','Page','ErrorPage','RedirectorPage','VirtualPage')
  character set utf8 collate utf8_general_ci default 'SiteTree')
• Field SiteTree_versions.ClassName: changed to enum('SiteTree','Page','ContentPage','ErrorPage','RedirectorPage','VirtualPage')
  character set utf8 collate utf8_general_ci default 'SiteTree' (from enum('SiteTree','Page','ErrorPage','RedirectorPage','VirtualPage')
  character set utf8 collate utf8_general_ci default 'SiteTree')
• Email_BounceRecord
• QueuedEmail
• File
• Group
• LoginAttempt
• Member
• MemberPassword
• Permission
• PermissionRole
• PermissionRoleCode
• Widget
• WidgetArea
• Page
• ContentPage
• ErrorPage
• Index ErrorPage_versions.RecordID_Version: changed to unique (RecordID, Version) (from unique (RecordID,Version))
• RedirectorPage
• Index RedirectorPage_versions.RecordID_Version: changed to unique (RecordID, Version) (from unique (RecordID,Version))
• VirtualPage
• Index VirtualPage_versions.RecordID_Version: changed to unique (RecordID, Version) (from unique (RecordID,Version))
• Image
• Folder
• Image_Cached

Creating database records

• PageComment
• SiteConfig
```

Now that we've added our own page, how do we use it? Go to the admin area of the CMS (if you already had it open before creating the page, reload it), then click on **Create** and you should see our new page, ready to be used. If you follow the UpperCamelCase convention spaces will be inserted for better readability, so it should look like this:



Simple, wasn't it? Let's use our content page: change the page type for all our pages, except for the error pages (you can leave them as they currently are), to **Content Page** (on the **Behaviour** tab of each page).

Using the Controller

Now that we've covered the overall structure, it's time to delve a little deeper into the Controller. Before we do that, let's just take a short look at some handy constants for handling paths gracefully.

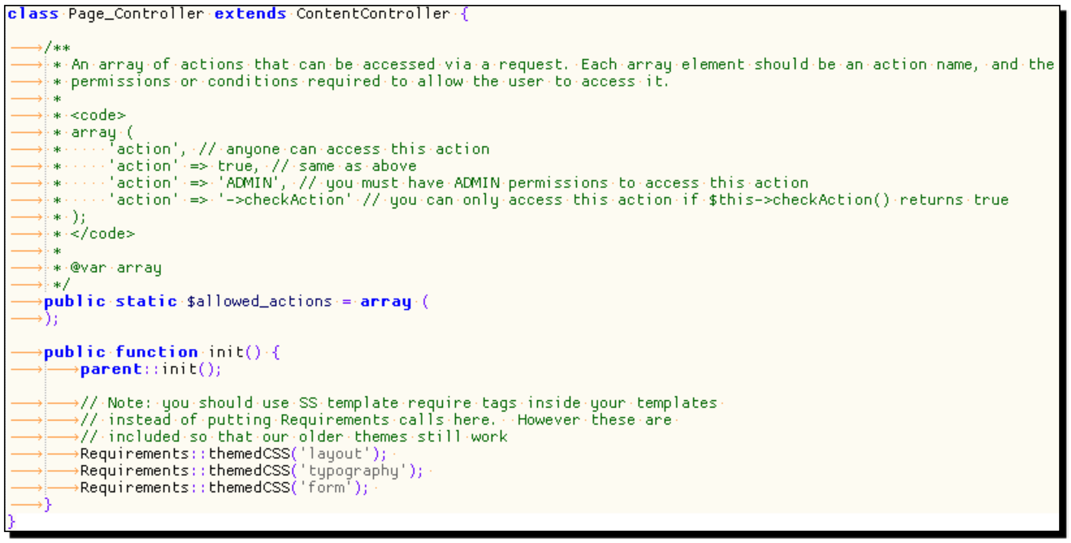
Path constants

All these constants return a path without trailing slashes. Use them whenever possible instead of hard-coding values. This will make it painless to upgrade SilverStripe later on, where paths might change.

`ASSETS_DIR`	Assets directory relative to the webroot: `assets`
`ASSETS_PATH`	Absolute path to the assets directory, for example: `/var/www/silverstripe3/assets` on Linux and Mac or `C:\Webserver\silverstripe3\assets` on Windows--to keep it simple in the future, we'll use Webserver/silverstripe3/assets for the absolute path
`BASE_PATH`	Absolute path to the site's root folder, for example, `Webserver/silverstripe3`
`BASE_URL`	The full URL to the site's base folder, for example, `http://localhost/silverstripe3` or `http://localhost` Equivalent to the template's `$BaseHref`
`CMS_DIR`	CMS directory relative to the webroot: `cms`
`CMS_PATH`	Absolute path to the CMS directory, for example, `Webserver/silverstripe3/cms`
`SAPPHIRE_DIR`	Relative path to the Sapphire directory: `sapphire`
`SAPPHIRE_PATH`	Absolute path to the Sapphire directory, you'll know it by now
`TEMP_FOLDER`	Absolute path to the temporary folder, for example, `silverstripe-cache`. For example, `Webserver/silverstripe3/silverstripe-cache`
`THEMES_DIR`	Relative path to the themes directory: `themes`
`THEMES_PATH`	Absolute path to the same directory
`THIRDPARTY_DIR`	Relative path to the third-party directory: `sapphire/thirdparty`
`THIRDPARTY_PATH`	Absolute path to the same directory

The default page class

Now it's time to get back to our application. Let's finally look at the content of `mysite/code/Page.php`. We're saving the Model for the next chapter, so we'll head to the Controller right away.



```
class Page_Controller extends ContentController {

	/**
	 * An array of actions that can be accessed via a request. Each array element should be an action name, and the
	 * permissions or conditions required to allow the user to access it.
	 *
	 * <code>
	 * array (
	 *     'action', // anyone can access this action
	 *     'action' => true, // same as above
	 *     'action' => 'ADMIN', // you must have ADMIN permissions to access this action
	 *     'action' => '->checkAction' // you can only access this action if $this->checkAction() returns true
	 * );
	 * </code>
	 *
	 * @var array
	 */
	public static $allowed_actions = array (
	);

	public function init() {
		parent::init();

		// Note: you should use SS template require tags inside your templates
		// instead of putting Requirements calls here.  However these are
		// included so that our older themes still work
		Requirements::themedCSS('layout');
		Requirements::themedCSS('typography');
		Requirements::themedCSS('form');
	}
}
```

Ignore the `$allowed_actions`. Security is something we'll concentrate on later. This leaves us with the following code in the class `Page_Controller` (without comments and newlines):

```
public function init(){
  parent::init();
  Requirements::themedCSS('layout');
  Requirements::themedCSS('typography');
  Requirements::themedCSS('form');
}
```

So there's a single method and it's special again. `init()` is called every time a page of this type is accessed. `parent::init()` calls the `init()` method of the inherited class, in this case `SiteTree`. Whenever you implement `init()` call `parent::init()` so you don't accidentally cut out some (inherited) initialization.

Setting up CSS

As `Page` is the base class for all other pages in the system, this method is actually called for every single page: ideal for handling things like CSS.

Including CSS files

The next three lines look very similar to `<% require themedCSS(layout) %>` in the template. They do exactly the same thing. This also solves the mystery of the second chapter's example: why was the CSS included even though we never referenced it in the View? Because it's actually added in the Controller. That means you can either add CSS in `themes/bar/templates/Page.ss` or in `mysite/code/Page.php`, the effect is the same.

However, the Controller variant adds one important feature: you can (optionally) add a second argument which sets the media attribute. This defines which displays the CSS should be used for.

Time for action – adding a print style

Let's create a CSS file specifically for printing (blanking out the background image and so on). The CSS so far should only be used for presentations and on screens—leaving us with this code:

```
Requirements::themedCSS('layout', 'screen,projection');
Requirements::themedCSS('typography', 'screen,projection');
Requirements::themedCSS('form', 'screen,projection');
Requirements::themedCSS('print', 'print');
```

What just happened?

This results in the following HTML:

```
<link rel="stylesheet" type="text/css" media="screen,projection"
href="http://localhost/themes/bar/css/layout.css?m=1284520662" />
<link rel="stylesheet" type="text/css" media="screen,projection"
href="http://localhost/themes/bar/css/typography.css?m=1284520688" />
<link rel="stylesheet" type="text/css" media="screen,projection"
href="http://localhost/themes/bar/css/form.css?m=1284520476" />
<link rel="stylesheet" type="text/css" media="print" href="http://
localhost/themes/bar/css/print.css?m=1284520757" />
```

By the way, including a CSS file from outside the `themes/` folder is accomplished by `Requirements::css('file', 'media');`.

> Note that when you're referencing a non-existent file, it's simply not added. No error message is generated either. This can be problematic when troubleshooting missing style sheet information.

Combining CSS files

We can even take this one step further. Too many HTTP requests slow down a website considerably. Loading these four CSS files alone isn't a big deal, but the fewer requests we need the better.

Time for action – reducing HTTP requests for CSS files

SilverStripe provides a mechanism to combine CSS files on the fly. As soon as one of the underlying files changes, a new combination is generated. The only problem is that combined files don't support the media attribute. But that's no real problem for us. We'll simply combine the three screen CSS files for everything and set up the print CSS file to override them where needed. So the new PHP code to accomplish that looks like this:

```
Requirements::themedCSS('print', 'print');
$theme = SSViewer::current_theme();
Requirements::combine_files('combined.css', array(
  THEMES_DIR . '/' . $theme . '/css/layout.css',
  THEMES_DIR . '/' . $theme . '/css/typography.css',
  THEMES_DIR . '/' . $theme . '/css/form.css',
));
```

Instead of using `Requirements::themedCSS()` we're providing the path in the form of: `THEMES_DIR . '/' . $themes . '/css/'`. Note that neither the themes' directory nor the currently selected theme are hard-coded—keeping our code clean and reusable.

What just happened?

The print CSS file is included like before. After that we fetch the current theme directory via `SSViewer::current_theme()` so we don't need to hard-code the path to the current theme directory. Finally, we define the name of the combined file and an array with all the CSS files that should be included.

Reload a page in the browser and look at the source. Nothing changed—try flushing. Still no change? You're not doing anything wrong. In development mode, files are not combined but included separately. So to see the results, change the mode in the `mysite/_config.php` file to `live`:

```
Director::set_environment_type('live');
```

> Once you have successfully tested the combined CSS file, switch the mode back to development. This will make debugging errors easier later on.

The HTML source in test mode should now reference the CSS files like this:

```
<link rel="stylesheet" type="text/css" media="print" href="http://
localhost/themes/bar/css/print.css?m=1284520757" />
<link rel="stylesheet" type="text/css" href="http://localhost/assets/_
combinedfiles/combined.css?m=1285714112" />
```

As you can see the combined file is automatically saved in the `assets/_combinedfiles/` folder. You can change that by putting a path before the filename in the string, but the default should be just fine.

[If your CSS is missing after switching to the combined file, make sure the web server is allowed to write to both the folder and the file itself.]

Adding custom CSS

If you need more control, for example generating dynamic CSS based on some non-static values, you can write page specific CSS to a page with `Requirements::customCSS('css', 'id');`. This is automatically added to your page's header. Replace the first argument with the custom CSS. `id` is used to uniquely identify a CSS snippet so it's not included multiple times on the same page. The ID is optional but generally it's a good idea to use it.

Pop quiz – duplication or not?

To illustrate that, let's take a look at the following example:

```
Requirements::css(THEMES_DIR . '/' . $theme . '/css/layout.css');
Requirements::css(THEMES_DIR . '/' . $theme . '/css/layout.css');
Requirements::customCss('.double { font-weight: bold; }');
Requirements::customCss('.double { font-weight: bold; }');
Requirements::customCss('.once { font-weight: bold; }', 'once');
Requirements::customCss('.once { font-weight: bold; }', 'once');
Requirements::customCss('.which { font-weight: bold; }', 'which');
Requirements::customCss('.which { font-weight: normal; }', 'which');
```

Assume you've added it to the `init()` method, which of the following statements is true?

1. The file `layout.css` is included twice. As there isn't an optional second ID argument for `Requirements.css()` there isn't anything we can do about it.
2. The CSS class `.double` is added twice in the pager's header section as it doesn't have a unique identifier.

3. The CSS class `.once` is added once in the pager's header section, as the unique identifier avoids the repetition.
4. The CSS class `.which` has a bold font weight as the second entry is ignored.

Of course, you can always add a `<style>` block in your template and this might also be a cleaner approach than putting layout information into the Controller. However, there are situations where it's harder to accomplish the same functionality in the View than relying on this approach.

Taking care of Internet Explorer

Internet Explorer, in particular its older versions, is often the problem child of web developers. On our page some of the nice CSS3 effects are ignored by that browser, such as the gradient background. However there are some proprietary tags to achieve the same effect.

Time for action – add your own head tags

We can set them up with Internet Explorer's conditional statements:

```
Requirements::insertHeadTags('
  <!--[if lt IE 9]>
    <style type="text/css">
      .transparent {
        background: transparent;
        -ms-filter: "progid:DXImageTransform.Microsoft.
                gradient(startColorstr=#44FFFFFF,
                          endColorstr=#B2FFFFFF)"; /* IE8 */
        filter: progid:DXImageTransform.Microsoft.
                  gradient(startColorstr=#44FFFFFF,
                            endColorstr=#B2FFFFFF); /* IE6 & 7 */
        zoom: 1;
      }
    </style>
  <![endif]-->
', 'IE-styling');
```

What just happened?

The code adds the first string, the conditional HTML, to the page's header. This technique is not limited to CSS, you can also include some custom meta tags or whatever you see fit. The second string is the ID of the snippet to avoid duplicate inclusions.

Alternatively, you can also detect the user agent on the server-side and only add browser-specific tags if the targeted browser is actually used.

Setting up JavaScript

JavaScript and CSS are often combined to create nifty effects and stylings. To keep things simple, they are included in SilverStripe much the same way. The equivalent for `Requirements::css()` is `Requirements::javascript('file')`—obviously there is no media attribute.

Being performance conscious, SilverStripe will add JavaScript to your page as late as possible, generally just above the `</body>` tag. This is the preferred method but in some cases it might break something. You can force SilverStripe to add JavaScript in the header region of your page, but only do that if it's really necessary: `Requirements::set_write_js_to_body(false);`

Including Google Analytics

Google Analytics is the de-facto standard for tracking users. Our bar page will be no exception.

Time for action – adding custom JavaScript in the Controller

We can include the necessary JavaScript code in a template file, but if you need a more versatile approach, you can also add JavaScript in the Controller:

```
if(Director::isLive()){
  Requirements::customScript("
    var _gaq = _gaq || [];
    _gaq.push(['_setAccount', 'UA-XXXXXXXX-X']);
    _gaq.push(['_trackPageview']);
    (function(){
      var ga = document.createElement('script');
      ga.type = 'text/javascript'; ga.async = true;
      ga.src = ('https:' == document.location.protocol ?
          'https://ssl' : 'http://www') +
               '.google-analytics.com/ga.js';
      var s = document.getElementsByTagName('script')[0];
              s.parentNode.insertBefore(ga, s);
    })();
  ", 'google-analytics');
}
```

What just happened?

We're only adding the custom JavaScript (`Requirements::customScript()`) if the page is in live mode. This ensures that we are not fudging the statistics during development. The second string is the `id` again, protecting against duplicate inclusions.

Blocking files

While adding CSS and JavaScript is the more important part, it can also be handy to remove (automatically included) files. This is done with `Requirements::block('file')`.

Time for action – removing JavaScript in the Controller

The following JavaScript files are included by default. As we're not using them, let's remove them and make our pages a little more light-weight:

```
Requirements::block(THIRDPARTY_DIR . '/prototype/prototype.js');
Requirements::block(THIRDPARTY_DIR . '/behaviour/behaviour.js');
Requirements::block(SAPPHIRE_DIR . '/javascript/prototype_
improvements.js');
Requirements::block(SAPPHIRE_DIR . '/javascript/
ConfirmedPasswordField.js');
Requirements::block(SAPPHIRE_DIR . '/javascript/ImageFormAction.js');
Requirements::block(THIRDPARTY_DIR . '/jquery/jquery.js');
Requirements::javascript(THIRDPARTY_DIR . '/jquery/jquery-packed.js');
```

What just happened?

By default, SilverStripe uses Prototype as its JavaScript library of choice. There is a push to replace it with jQuery but currently they are both used. In the frontend we don't need Prototype at all. So we get rid of it along with some other custom JavaScript that we don't need. For speeding things up we'll use the compressed jQuery library instead of the regular one. That's what the last two lines do.

[There are some modules relying on Prototype. Take care when blocking in case you break some dependency.]

What about the external JavaScript file from the previous chapter we included for older versions of Internet Explorer? While it's tempting to try our new tools on it as well, better leave that as it is. For the script to work, it must be loaded before it's actually used, so we need it in the head section of the page. By default, any script added in the Controller is added to the page's bottom. This is generally good for loading pages as fast as possible, but in some specific cases might break functionality.

Where to include CSS and JavaScript

You've seen that we can include CSS and JavaScript both in the View and the Controller. The following bulletin points list some things to consider when choosing between the two methods:

- For the final output it doesn't make a difference.
- During development it's a bit of trade-off between control and clean separation of the layers. Putting layout information and client-side scripts into the Controller is not the intention of MVC.
- From a practical point of view, you don't have much choice if you need some logic (like checking for the current mode) or a feature that is only available in the Controller (the media attribute for example). So we'll definitely have to put these cases into the Controller.
- If you're doing server- and client-side validation of user inputs (we'll come to that in a later chapter) the two will be very similar. Splitting them up might make it harder to keep them synchronized, but this is a question of personal preference.
- You can't use path constants in the View, like `THIRDPARTY_DIR`.
- If you feel more comfortable relying on HTML's capabilities for including CSS and JavaScript, you can exclusively use this approach. SilverStripe doesn't enforce one way over another.
- For the rest it's up to you. In the examples, we'll use both approaches, depending on their specific use case. This should be acceptable as our page will always be rendered in human-readable form. If your use case might require XML output to some other system, you may need to take a different approach.

Coding conventions

Before hacking together another feature, we'll revisit coding conventions. The earlier you start using them, the more natural it will become.

Coding

On the code level there are just some recommendations you should follow. They have hardly any impact on the functionality, but mainly help with sticking to a consistent style.

Indentation

Use tabs, one per nesting level, instead of spaces for indentation.

Curly braces

Keep opening braces on the same line as the statement, as we've done it in the previous examples. It should be:

```
if(Director::isLive()){
```

instead of:

```
if(Director::isLive())
  {
```

Check before looping

Always check a variable before using it in a loop. This will help you prevent quite some unnecessary errors with inexistant or null variables:

```
if($elements){
  foreach($elements as $element){

  }
}
```

Keeping the Controller clean

Don't use HTML in the Controller. Only fetch and process the string itself and keep the markup strictly within the template.

Comments

Provide DocBlock comments for each class, method and when needed elsewhere—right before the targeted block. DocBlock comments are used in many languages, PHP for example. Using `http://www.phpdoc.org`, documentation can be generated automatically. SilverStripe does it for the core classes and you can and should do it for your own code as well. Visit the page if you're interested in the finer details as we'll be only able to touch on the subject.

Use a description for each comment and insert a newline before annotations. Classes should be annotated with `@package`. Methods should include `@param` and `@return` (where applicable). `@todo` and `@see` can be used where necessary. An example for a class is as follows:

```
/**
 * Define the base page.
 * Optionally a second description line or even more.
 *
 * @package mysite
 */
class Page extends SiteTree {
```

Two examples for methods are:

```
/**
 * Initializer, used on all pages.
 */
public function init(){
}

/**
 * Provide a random string to be added to a page.
 *
 * @return string The random string.
 * @todo Add the logic.
 */
protected function Random(){
  return '';
}
```

As you can see, a minimal DocBlock comment has three lines, each starting with an asterisk except for the first one—this is necessary for the documenter to pick it up. The description is one or more lines long.

If further elements for commenting are available, insert a blank line and then add the annotation(s):

- `@package` is followed by a single string, the desired package. The base folder name is generally a good choice.
- `@param` describes the parameters of a method. First the variable type is provided (`string`, `int`, `array`, and so on) followed by a description.
- `@return` describes the return value. The structure is the same as with `@param`.
- `@todo` is followed by a string describing what should actually be done.
- `@see` can be used for referencing methods, classes, defines, and more, for example: `@see SomeClass::MyMethod()`.

Writing meaningful comments is pretty hard. Without going into too much detail, focus on general purpose, inputs, and outputs while keeping comments concise and up to date.

Take the two method comments above as examples for good comments. A bad comment can, for example, be:

```
/**
 * Finally figured this stupid thing out, lol.
 * Assign the value to the variable and output it.
 */
public function doIt(){
  $variable = 'foobar';
  echo $variable;
}
```

The first line is totally redundant. The second line of the comment isn't really informative as well. Any programmer knows how to assign a variable and how to output it, this information is pointless. However, the purpose is left in the dark—what are we actually trying to achieve and why are we specifically using this output?

The only thing worse than bad comments is bad code: don't document it—fix it and then document it!

Methods

Functions take a fundamental role in your pages' functionality. Not only does this section introduce the coding conventions, but also the basic concepts themselves. We'll dig deeper into them over the course of this and also the next chapters.

Sticking to these rules is more important than with the general coding rules, as SilverStripe relies on certain standards, to create consistent URLs for example.

Static methods

Static methods are not bound to a specific object, but exist once for the whole class. They are often used for returning some value that is global to the class.

Static functions should be written in lowercase_with_underscores: `static function get_global_value(){`.

You will call this function via `Class::get_global_value();`.

Action handler methods

Methods can be used for mapping to a specific URL. For this use they should be in lowercase alphanumeric: `function showurl(){`.

If you create the above function in a page's controller and create a page of this type under the URL `/testpage`, `/testpage/showurl` will map to this function. This kind of function has no parameters.

[This is an important concept—we'll hear more about it later on.]

Template methods

Functions can also be used for fetching values in the template. They should be in UpperCamelCase and can optionally be preceded by a "get": either `function SomeValue(){` or `function getSomeValue(){`.

Both of them can be accessed in the template via the placeholder `$SomeValue`. These functions generally don't have any parameters.

Additionally, they don't have any side effects, as the return value is cached after the first access. Let's take a look at an example which tries to increment an integer:

```
private increment = 0;

function Incrementor(){
  $this->increment += 1;
  return $this->increment;
}
```

If you now call the method four times in the View, this will output `1 1 1 1` and not `1 2 3 4`:

```
$Incrementor
$Incrementor
$Incrementor
$Incrementor
```

Object methods

Object methods are used for internal processing, such as in a helper function. They should be written in lowerCamelCase: `function calculateAverage()`.

This method can be accessed in your Controller via `$this->calculateAverage()`. Or `$theObject->calculateAverage();` from another object where `$theObject` is an instance of the class that provides this method.

Variables

Variables are easy in comparison to methods:

- Object variables, being available in the whole object and not just one method, should be in `$this->lowerCamelCase`.
- Static variables, having a single instance for the whole class, should be in `self::$lowercase_with_underscores`.

Element order

Keeping elements in a specific order makes finding them easier. Unused elements can of course be left out.

First use the base class, then the page class and then any helper classes, in alphabetical order. Within a class stick to this order:

1. Static variables
2. Object variables
3. Static methods
4. Predefined SilverStripe methods, for example, `init()`
5. Action handler methods
6. Template methods
7. Object methods

Spam protecting e-mail addresses

We've already included an e-mail address on our page in the second chapter. Spam bots can, however, easily fetch it and then hammer us with unwanted mails. So let's fix that. There are many solutions, from generating images to various JavaScript based approaches.

Note that our approach isn't foolproof. However, it's still an improvement and nicely illustrates how seamless Controller, View, and CMS can work together.

We'll use a simple JavaScript solution which transforms `contact at bar dot com` into `contact@bar.com`. If you don't have JavaScript enabled you won't see the pretty e-mail address, but nothing else will break.

Unobtrusive JavaScript

JavaScript should always degrade gracefully. Provide additional features and effects, but don't break the basic functionality of your site without scripting. We'll take care in our examples to stick to this principle.

Controller

So what do we need to do? Add some JavaScript in the `init()` method of `mysite/code/Page.php` as we want to be able to use it anywhere on our page:

```
Requirements::customScript("
  $(document).ready(function(){
    $('span.mail').each(function(){
      $(this).replaceWith($(this).text().replace(/ at /,
        '@').replace(/ dot /g, '.'));
    });
  });
", 'mail-protect');
```

This snippet relies on jQuery. If you don't use jQuery, you can apply this same logic using your preferred JavaScript framework. Basically, it just looks for every `span` tag with a class of `mail`. Inside of that it replaces "` at `" (note the additional spaces) with `@` and "` dot `" with a dot (the sign and not the word). Finally, we're adding a unique ID so that our snippet is only included once in on each page.

You might also want to consider putting the JavaScript block into a dedicated file as it doesn't really fit into the Controller. The first time your page is being loaded, this requires an additional HTTP request. On subsequent pages the browser should normally have cached the file and you don't need to resend its content in each page's HTML body.

We could have built a solution that outputs a clickable link for sending e-mails directly, but as we'll build a form-based solution later anyway it's not really necessary.

Next we need to add that CSS class to our templates and use it in the CMS' content field.

Template

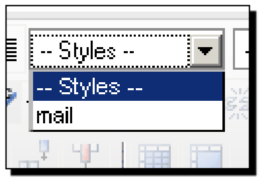
In your `themes/bar/templates/Includes/BasicInfo.ss` replace `contact@bar.com` with `<span class="mail">contact at bar dot com</span>`. Reload your page, don't forget flushing, and you shouldn't see a difference. Only if you disable JavaScript you'll see the underlying string. But that should be acceptable as most users have JavaScript enabled and the e-mail address can still be easily understood by humans. Spam bots, however, will have a harder time picking it up.

CMS

Add the following lines to your theme's `typography.css` file. Yes, there is no styling included, we just register our custom CSS class.

```
span.mail {
}
```

After reloading the CMS you should now have **mail** in the styles dropdown.



Now in the CMS you can simply add **me at mail dot com** in the content field, apply the **mail** styling and it will be rendered as **me@mail.com** in the frontend (if JavaScript is enabled). So the content editor doesn't need to work on the HTML source.

URL variables and parameters

After checking out the Controller itself, let's take a better look at the available URL variables and parameters. We'll save Model-specific parameters for the next chapter as they wouldn't make too much sense just yet.

Activating changes

In the second chapter we've already met `?flush=all` for clearing the cache and rebuilding the templates. In this chapter we've discovered `/dev/build`.

As you'll often introduce Model and Controller at the same time as a new View, you can use the two together: `/dev/build?flush=all`. But flushing is even more powerful. With the option of `1` you call the action on the current page and template used for displaying it only. By using `all` instead you're doing it for the entire system. So after introducing changes you'll generally just go to `/dev/build?flush=all`.

There are two more options you can specifically use on `/dev/build`:

- `/dev/build?quiet=1` won't report the actual actions done.
- `/dev/build?dont_populate=1` will just do the **Creating database tables** part but leave out the **Creating database records**. You can define default database records alongside the tables' structure. This parameter will only execute the latter part while leaving out the former one.

Just like with any other URL parameter you can use more than one. Simply concatenate them with an `&`, for example, `/dev/build?flush=all&dont_populate=1`.

If you have a lot of unpublished pages in your system, it's quite cumbersome to publish them all by hand. For example, if you're just doing a relaunch of a site and you need to put dozens of pages online at the same time. Luckily there's a handy tool just for that case: go to the URL `/admin/publishall/` to publish all pages at once.

Debugging

In the previous chapter, we've already seen `?isDev=1&showtemplate=1` for debugging templates. What does the statement really do?

Not seeing the output?

Our current template is not ideal for debugging as the background image can be over the debug output. Either be sure to apply the right CSS styling (see `layout.css`) or view the page's source.

Template

First it forces the site to be in development mode for this request (`isDev=1`). Only in development do you get access to the template, otherwise nothing is displayed for security reasons. So if you're working on a live site, the first part of the code is really the prerequisite for displaying the actual template code with `showtemplate=1`. For sites in development mode, `?showtemplate=1` will be sufficient but the additional `isDev=1` doesn't hurt.

With `isTest=1` you can force a site to be in the test mode, overriding the configuration.

Controller

The manifest caches information about all available classes, class hierarchy, tables in the database, and templates in order to speed up the system. For showing the current manifest, use `?debugmanifest=1`. You can rebuild it via `/dev/build`.

Showing all available methods on a page is done with `?debugmethods=1`. This can be handy for discovering functionality as well as extending existing functionality.

Wanting to know which class is currently used for the page and what the extended one is? Use `?debugfailover=1`, which will output:

Debug (line 177 of ViewableData.php): ContentPage_Controller created with a failover class of ContentPage

Request

With `?debug_request=1` you can show all classes involved in creating the current page, starting with the initial HTTP request to a controller and ending with the rendering of a template. This is handy for knowing what is really involved in the execution of a page, especially when you're tracking down errors.

In live mode the `?isDev=1` is required when trying to look at the request.

Debug

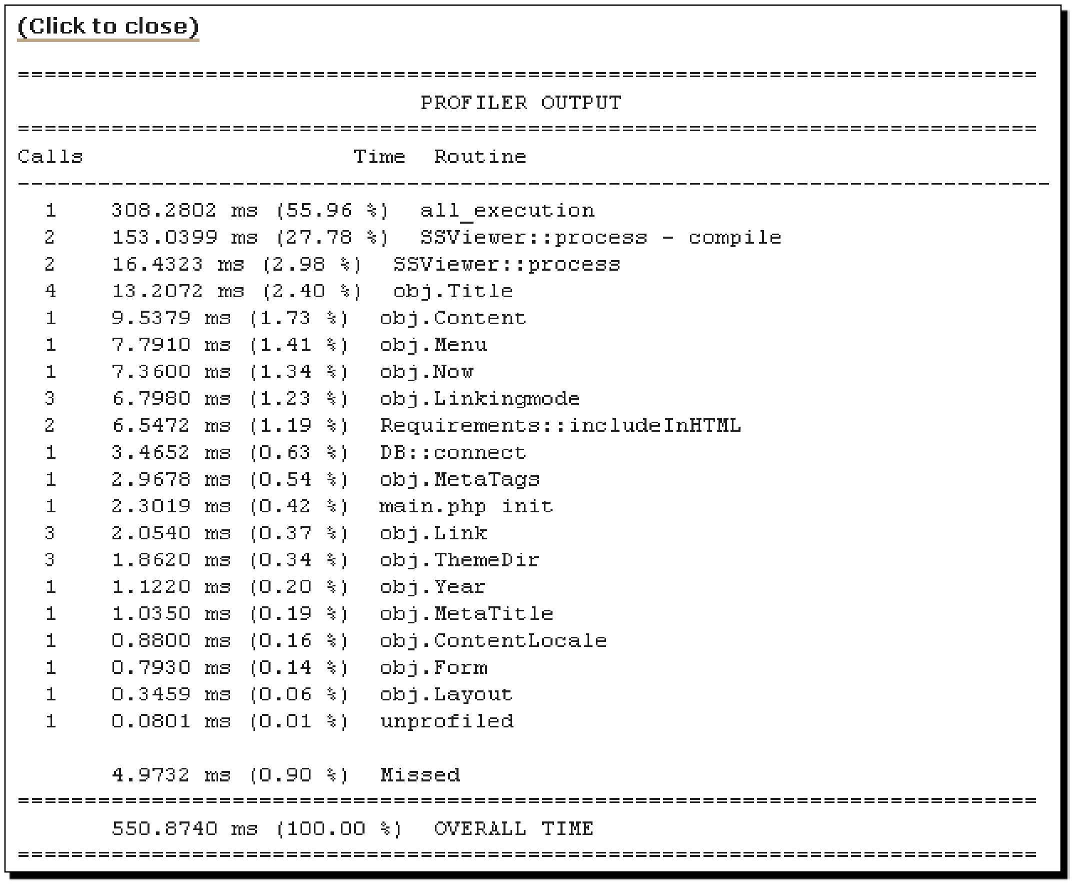
`?debug=1` can be used for debugging information about the Controller operation. It displays every URL configured in the system. This can be helpful when trying to resolve a conflict. And don't forget the `?isDev=1` when required by the environment.

Performance

If your page is not as fast as you'd have hoped, but you have no idea why or where to start looking, SilverStripe provides some useful tools for getting you started quickly.

Note that we won't cover JavaScript parameters as client-side scripting is best done via a browser plug-in like Firefox's Firebug.

`?debug_memory=1`	Shows the peak memory in bytes used for displaying the current page. Helpful for getting a feeling of the server-side resource usage.
`?debug_profile=1`	Use this if your pages are slow. It will provide a detailed output on how long the processing of the page took and also the specific parts. This really pins down bottlenecks in your code.
`?debug_profile=1&profile_trace=1`	Adds a full stack trace to the previous output and can only be used in combination with it. This is good for seeing in which order different classes are called and how they interact.



(Click to close)

```
============================================================================
                              PROFILER OUTPUT
============================================================================
Calls                         Time  Routine
-----------------------------------------------------------------------------
  1     308.2802 ms (55.96 %)  all_execution
  2     153.0399 ms (27.78 %)  SSViewer::process - compile
  2     16.4323 ms (2.98 %)  SSViewer::process
  4     13.2072 ms (2.40 %)  obj.Title
  1     9.5379 ms (1.73 %)  obj.Content
  1     7.7910 ms (1.41 %)  obj.Menu
  1     7.3600 ms (1.34 %)  obj.Now
  3     6.7980 ms (1.23 %)  obj.Linkingmode
  2     6.5472 ms (1.19 %)  Requirements::includeInHTML
  1     3.4652 ms (0.63 %)  DB::connect
  1     2.9678 ms (0.54 %)  obj.MetaTags
  1     2.3019 ms (0.42 %)  main.php init
  3     2.0540 ms (0.37 %)  obj.Link
  3     1.8620 ms (0.34 %)  obj.ThemeDir
  1     1.1220 ms (0.20 %)  obj.Year
  1     1.0350 ms (0.19 %)  obj.MetaTitle
  1     0.8800 ms (0.16 %)  obj.ContentLocale
  1     0.7930 ms (0.14 %)  obj.Form
  1     0.3459 ms (0.06 %)  obj.Layout
  1     0.0801 ms (0.01 %)  unprofiled

        4.9732 ms (0.90 %)  Missed
============================================================================
        550.8740 ms (100.00 %)  OVERALL TIME
============================================================================
```

While SilverStripe's debugging features are good for getting a general overview, there are more solid and in-depth ways of profiling your application:

- Xdebug (`http://xdebug.org`) adds powerful debugging features to PHP as well as profiling
- XHProf (`http://pecl.php.net/package/xhprof/`) is another profiler, originally developed at Facebook

Debugging the code

The most important step for debugging a problem is being in development mode. This sounds trivial but is a constant source of frustration.

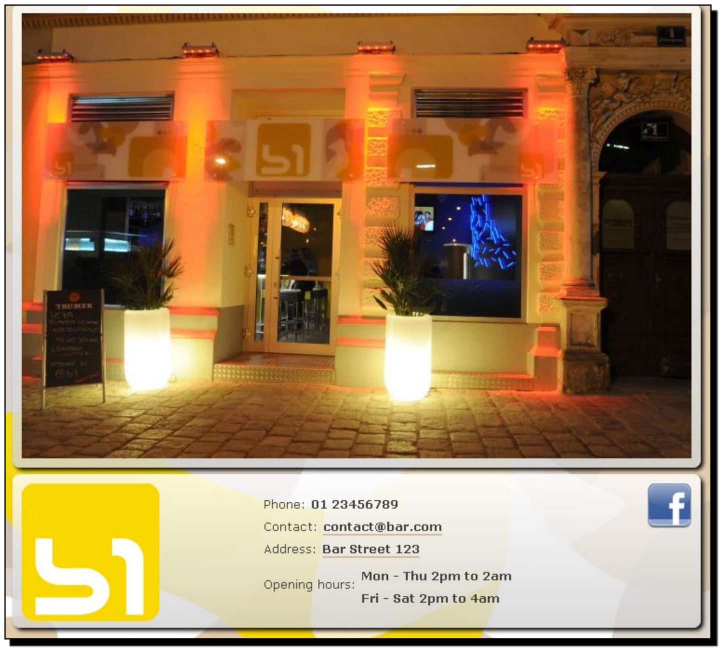
Besides URL parameters there are some other helpful methods for debugging PHP code:

- `Debug::show($variable)`: SilverStripe's `print_r()` on steroids. It displays the class, method, and line number where used and styled with a bit of HTML.
- `Debug::message('My message')`: Like the above but for displaying a message.
- `SS_Backtrace::backtrace()`: Prints a call trace for the current page. Shows which Controllers and Models are actively involved in the creation of this page. Processing continues after displaying the call's stack.
- `user_error('My message', E_USER_ERROR)`: Like the above but stops the execution after the call and adds a custom message.

If you're having trouble with a specific file, add a few `Debug::show()` and `Debug::message()` at key places to see what's really going on.

Adding an Intro page

Our bar page still looks a bit simple; we need some action. Let's add some auto-rotating images. We've found a nice example that doesn't include any huge libraries, see `http://www.alohatechsupport.net/webdesignmaui/maui-web-site-design/easy_jquery_auto_image_rotator.html`. Now it's your job to set this up, you already know everything you need. The result should look like this, including a fancy fade between images, which is difficult to show in print:


Phone: 01 23456789
Contact: contact@bar.com
Address: Bar Street 123
Opening hours: Mon - Thu 2pm to 2am
Fri - Sat 2pm to 4am

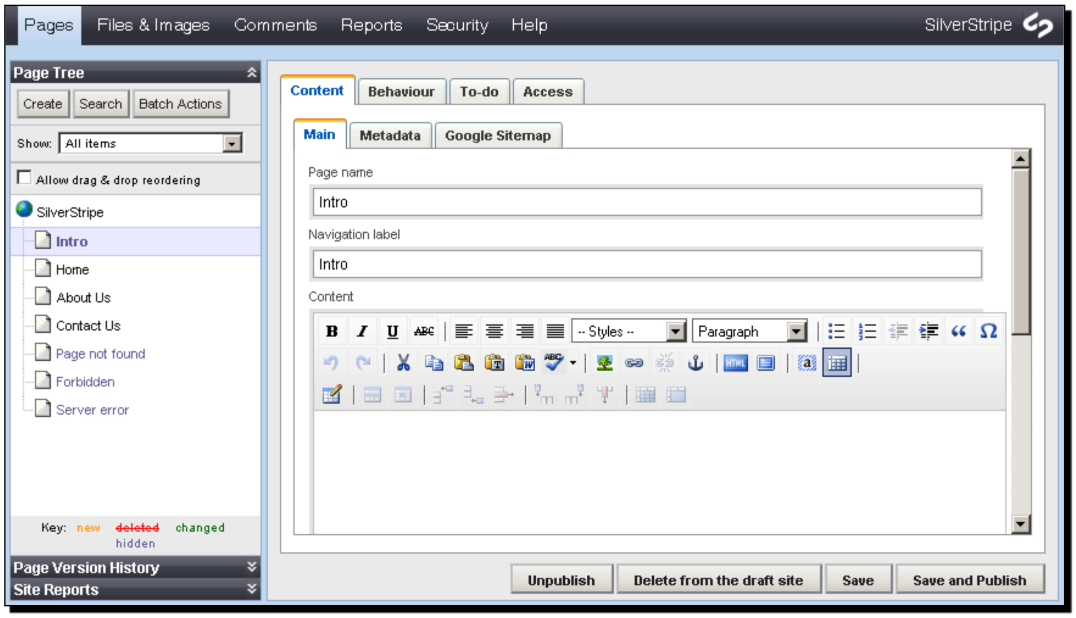
Time for action – add an Intro page

Let's start with the following steps:

1. Create a new page type just like `ContentPage.php`. Call it `IntroPage.php`, since it's exclusively used for our intro.
2. In your custom page, use the `init()` function to include the necessary JavaScript. We can also create a dedicated file for it, but that will be another HTTP request that we can easily avoid. Additionally, because this script is only needed on the intro, caching it wouldn't provide any benefit for other pages.
3. Put the images in the `assets/Uploads/` folder as we'll later enable the content editors to add and remove images. For keeping things organized, create a new folder where you only store the intro images. So `assets/Uploads/intro/` will be a logical choice for the folder name.
4. Make sure the images have equal dimensions, for example, 720 pixels wide. We'll later see how SilverStripe can resize images for us, but we'll need the Model for that.
5. Add a new template file. Use the correct filename!
6. Build the layout of your intro page. Now you can see why we put so little information into the base `Page.ss`—only what is used on all pages, which enables us to use two very distinct page layouts inside the `themes/bar/templates/Layout/` folder.
7. Also use the `BasicInfo.ss` include. We might style it a little different, but the basic information behind it is the same.
8. Link the page's background and the rotating images to `/start`.

On the code level we're done. Synchronize the changes with the database and log in to the CMS backend:

9. In the CMS create a newly added **Intro Page**. There's nothing to add on this page right now.



Page name: Intro

Navigation label: Intro

Content

10. Switch to the **Home** page and change its URL to `start`.

11. For the **Intro** change the URL to `home`.

12. Remove the **Intro** from the Menu. We'll put a link next to the copyright notice, but we don't need it in the menu.

That's everything you need to do to add our fancy Intro page. If you got stuck somewhere, take a look at the code provided in the following sections. This should help you getting back on the right track.

The contents of the `mysite/code/IntroPage.php` file is as follows:

```
<?php

class IntroPage extends Page {
}

class IntroPage_Controller extends Page_Controller {
  public function init(){
    parent::init();

    Requirements::customScript("
      function theRotator(){
        $('#rotator ul li').css({opacity: 0.0});
        $('#rotator ul li:first').css({opacity: 1.0});
        setInterval('rotate()', 3000);
      }
      function rotate(){
        var current = ($('#rotator ul li.show') ?
            $('#rotator ul li.show') : $('#rotator ul li:first'));
        var next = ((current.next().length) ?
                   ((current.next().hasClass('show')) ?
                     $('#rotator ul li:first') : current.next()) :
                     $('#rotator ul li:first'));
        next.css({opacity: 0.0}).addClass('show').
                    animate({opacity: 1.0}, 1000);
        current.animate({opacity: 0.0}, 1000).removeClass('show');
      };
      $(document).ready(function(){
        theRotator();
        $('#rotator').fadeIn(1000);
        $('#rotator ul li').fadeIn(1000); /* Tweak for IE */
      });
    ");
  }

}
```

The template `themes/bar/templates/Layout/IntroPage.ss` is as follows:

```
<div><a href="start"><img src="$ThemeDir/images/background.jpg"
alt="Background" id="background"/></a></div>

<figure id="rotator">
  <ul>
    <li class="show">
      <a href="start">
       <img src="assets/Uploads/intro/bar_start-1.jpg"
        alt="Intro image 1" class="rounded transparent-nonie shadow"/>
      </a>
    </li>
    <li>
      <a href="start">
```

```
          <img src="assets/Uploads/intro/bar_start-2.jpg"
           alt="Intro Bild 4" class="rounded transparent-nonie shadow"/>
        </a>
      </li>
      <li>
        <a href="start">
          <img src="assets/Uploads/intro/bar_start-3.jpg"
           alt="Intro Bild 3" class="rounded transparent-nonie shadow"/>
        </a>
      </li>
      <li>
        <a href="start">
          <img src="assets/Uploads/intro/bar_start-4.jpg"
           alt="Intro Bild 4" class="rounded transparent-nonie shadow"/>
        </a>
      </li>
    </ul>
  </figure>

  <aside id="bottom" class="transparent rounded shadow">
    <% include BasicInfo %>
  </aside>
```

Additionally, you'll need some styling information—otherwise the intro page won't be displayed correctly. But you don't need to type it in manually, you can simply copy it from the book's code repository, provided at `http://www.packtpub.com/support`. We're just providing it here for the sake of completeness or in case you're not in front of a computer right now.

Add the following code to `themes/bar/css/layout.css`. As it's general styling information for the page and not related to a CMS field, it should be located in the layout file:

```
#rotator {
    position: relative;
    top: 5px;
    left: 50%;
    height: 470px;
    margin-left: -400px;
    width: 800px;
}
#rotator ul li {
    float: left;
    position: absolute;
```

```
        list-style: none;
    }
    #rotator ul li img {
        padding: 10px;
    }
    #rotator ul li.show {
        z-index: 500;
    }
    #rotator a {
        text-decoration: none;
        border: 0;
    }
```

Later we'll make some changes to this code so that content editors can take full control over the intro page. So far, it's rather static and cumbersome, but it's mainly for showing how to implement such a feature into your SilverStripe page and how to organise the content.

What just happened?

There shouldn't be any real surprises in this example, but let's recap the general workflow for creating a new page type:

- First you need to set up the code, specifically the (currently empty) Model and Controller. In our specific example we're setting up all the required information via the Controller's `init()` method.
- Add a template file and styling information as required.
- Rebuild the database so that your changes are picked up.
- Reload the CMS, create a page based on our new page type and provide all the required information.

Pop quiz – basic principles

What are the three most important principles of SilverStripe and what does each of them achieve?

Summary

In this chapter we've covered the Controller role. While we've already built in some nice features there's a lot more to come. We've also laid the groundwork for the following chapters.

Specifically, we covered:

- MVC and the general architecture used by the framework
- How to create our own page types
- Taking the first steps on how to create some logic for our pages
- How to use the URL tools

We've already covered View and Controller, but we are totally missing the Model. Once we come to that we'll really get things going!

4
Storing and Retrieving Information

Having covered the View and Controller roles in the previous chapter, we'll now tackle the remaining one: the Model. In this chapter we'll take a look at how SilverStripe "sees" and interacts with the database. While there are many finer details (of which we'll cover the most important ones), this concludes the overview of SilverStripe in general.

In this chapter we will:

- Explore how to access the database via the Model
- See how to extend the Model to suit our own needs
- Additionally see what else we can do in the Model and how all of this enriches our site

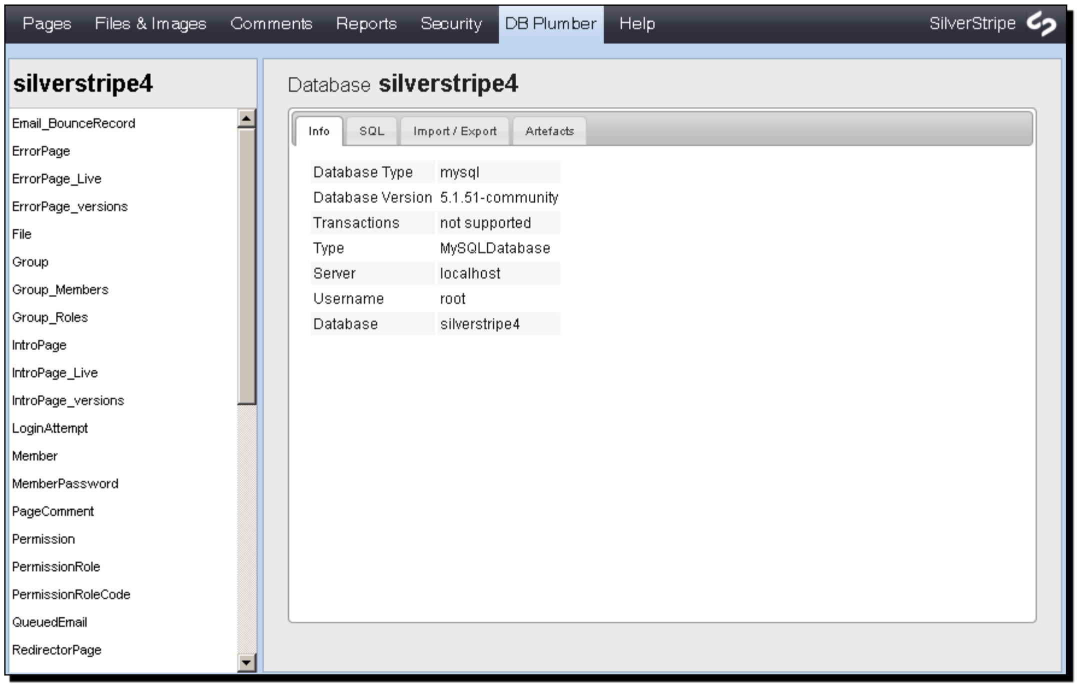
At the end of this chapter you'll know everything needed to build a basic site, so let's start, shall we?

DBPlumber

Before starting anything complicated we should take a look at the database, but for doing that we need another tool.

Your new best friend: DBPlumber

Many of you will be familiar with phpMyAdmin, `http://www.phpmyadmin.net`, for managing MySQL databases. Some will swear on MySQL Workbench, `http://wb.mysql.com`, the command-line client, or some other tool. And then there's the a SilverStripe-specific tool called DBPlumber.



Database Type	mysql
Database Version	5.1.51-community
Transactions	not supported
Type	MySQLDatabase
Server	localhost
Username	root
Database	silverstripe4

Why do we need another tool?

- It's a module for SilverStripe and integrates itself into the CMS. Therefore, you don't need to open up another program and log into that. Once you're logged into the CMS you're ready to go.
- You don't need to maintain and configure another tool. DBPlumber simply uses the already existing configuration file.
- It fits well into the ecosystem. It's simple and enables you to do all the common tasks you should need without being complicated. This includes:
    - Easily inserting, updating, or deleting data
    - Backing up individual, multiple, or all tables in the database
    - Importing backups
    - Running custom SQL statements

- It works across all supported databases—one tool to rule them all.

Note that you'll need admin rights to access DBPlumber. But that makes sense—you wouldn't want your content editors to mess up your database now, would you!

Time for action – installing DBPlumber

DBPlumber installs just like any other module. Let's quickly go through the process, so that you can easily do it on your own in the future:

1. Back up your database!

   Okay, in this instance it's kind of difficult as we're just about to install the tool required for it. But for the future, always back up your database before making any changes to your installation by going to DBPlumber's **Import/Export** tab (see the previous screenshot) to export your data. Or use any other tool you're familiar with. It's a minute well invested, so just include it into your routine. Just this once we'll jump that step.

2. Get the code:

   Download the archive at `http://silverstripe.org/database-plumber-module/`. Unpack it and rename it to `dbplumber/`—this is vital or it won't work correctly. Always use the filename of the archive without the "-trunk-r" and revision number. Then copy the folder over to your site's base folder. `dbplumber/` should now reside just beside the folders `sapphire/`, `cms/`, or `googlesitemap/`.

3. Inside the DBPlumber module, you'll find a `_config.php` file as well as some other already known folders, like `code/`, `templates/`, and so on.

   Note that every module must at least contain a configuration file, otherwise SilverStripe won't pick it up.

4. Head over to our old friend `/dev/build?flush=all`. If you're neither in development mode nor logged in, you'll be prompted for your credentials now.

That's it, you're done. In the CMS interface you should now have a new entry in the top menu bar called **DB Plumber**. You can also access it directly via `/admin/dbplumber/`.

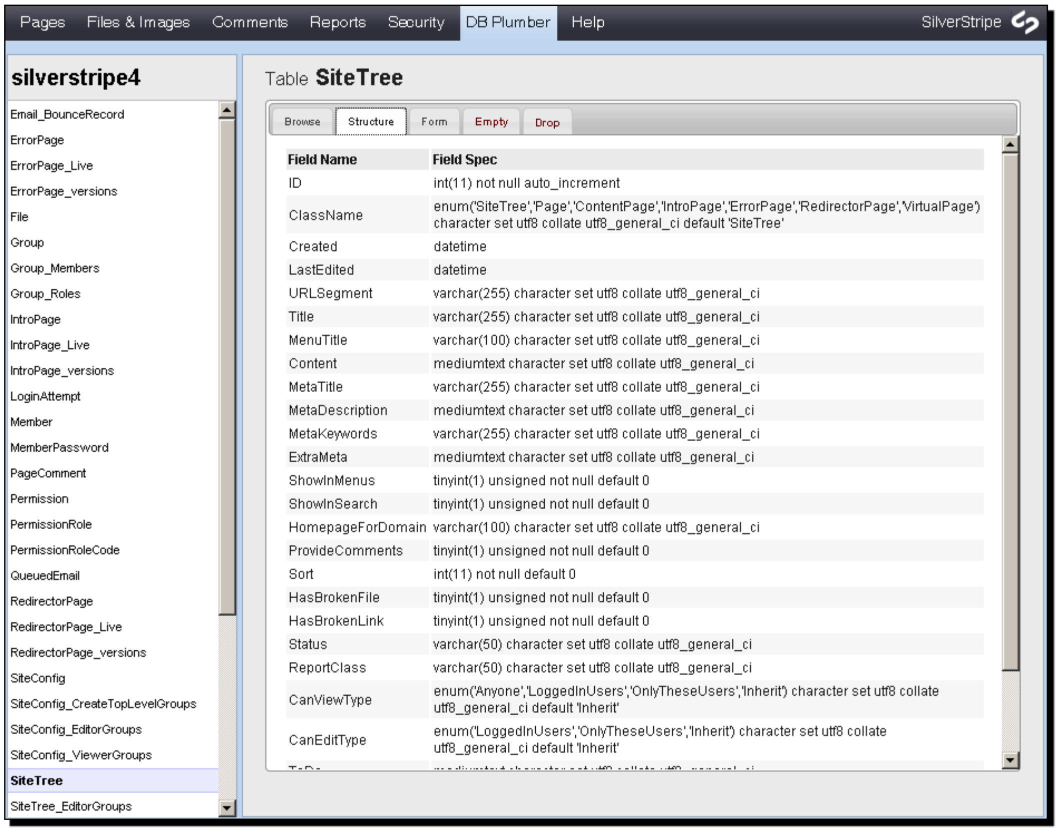
What just happened?

Once you've called `/dev/build`, SilverStripe's underlying framework Sapphire will search the base folder. If it finds any changes (added, removed, or updated files and folders), it will update the database accordingly. After finishing, any changes should have been applied and are now used in the frontend, backend, or both. As you might expect, DBPlumber will only have an impact on the backend.

We won't go over all the features of DBPlumber as it should be pretty self-explanatory. Instead, let's dive right into the database structure.

From a database's point of view

Open the **SiteTree** database table by selecting it from the left-hand panel. It contains all the pages of our site.



silverstripe4

Email_BounceRecord
ErrorPage
ErrorPage_Live
ErrorPage_versions
File
Group
Group_Members
Group_Roles
IntroPage
IntroPage_Live
IntroPage_versions
LoginAttempt
Member
MemberPassword
PageComment
Permission
PermissionRole
PermissionRoleCode
QueuedEmail
RedirectorPage
RedirectorPage_Live
RedirectorPage_versions
SiteConfig
SiteConfig_CreateTopLevelGroups
SiteConfig_EditorGroups
SiteConfig_ViewerGroups
SiteTree
SiteTree_EditorGroups

Table **SiteTree**

Field Name	Field Spec
ID	int(11) not null auto_increment
ClassName	enum('SiteTree','Page','ContentPage','IntroPage','ErrorPage','RedirectorPage','VirtualPage') character set utf8 collate utf8_general_ci default 'SiteTree'
Created	datetime
LastEdited	datetime
URLSegment	varchar(255) character set utf8 collate utf8_general_ci
Title	varchar(255) character set utf8 collate utf8_general_ci
MenuTitle	varchar(100) character set utf8 collate utf8_general_ci
Content	mediumtext character set utf8 collate utf8_general_ci
MetaTitle	varchar(255) character set utf8 collate utf8_general_ci
MetaDescription	mediumtext character set utf8 collate utf8_general_ci
MetaKeywords	varchar(255) character set utf8 collate utf8_general_ci
ExtraMeta	mediumtext character set utf8 collate utf8_general_ci
ShowInMenus	tinyint(1) unsigned not null default 0
ShowInSearch	tinyint(1) unsigned not null default 0
HomepageForDomain	varchar(100) character set utf8 collate utf8_general_ci
ProvideComments	tinyint(1) unsigned not null default 0
Sort	int(11) not null default 0
HasBrokenFile	tinyint(1) unsigned not null default 0
HasBrokenLink	tinyint(1) unsigned not null default 0
Status	varchar(50) character set utf8 collate utf8_general_ci
ReportClass	varchar(50) character set utf8 collate utf8_general_ci
CanViewType	enum('Anyone','LoggedInUsers','OnlyTheseUsers','Inherit') character set utf8 collate utf8_general_ci default 'Inherit'
CanEditType	enum('LoggedInUsers','OnlyTheseUsers','Inherit') character set utf8 collate utf8_general_ci default 'Inherit'

Every base table in a SilverStripe database has the first four properties; they are automagically created for every entry:

- **ID**: The unique ID of this record. The **SiteTree** table's ID 1 corresponds to `/admin/show/1`, the first page on your CMS' **Pages** tab. If you change back to the **Pages** tab from **DB Plumber** in the CMS and hover the mouse pointer over the **Page Tree** elements on the left-hand side, you can actually see the IDs in the URL. So while the ID is used internally SilverStripe uses human-readable URLs externally.

- **ClassName**: Defines which PHP classes map to this table. Sometimes it's just a single class, but for the **SiteTree** table this includes the **SiteTree** class and any class which extends it. So here we can find our **ContentPage** and **IntroPage** classes.
- **Created**: Timestamp when the record was created.
- **LastEdited**: Timestamp when the record was last edited, but you'd have guessed that anyway.

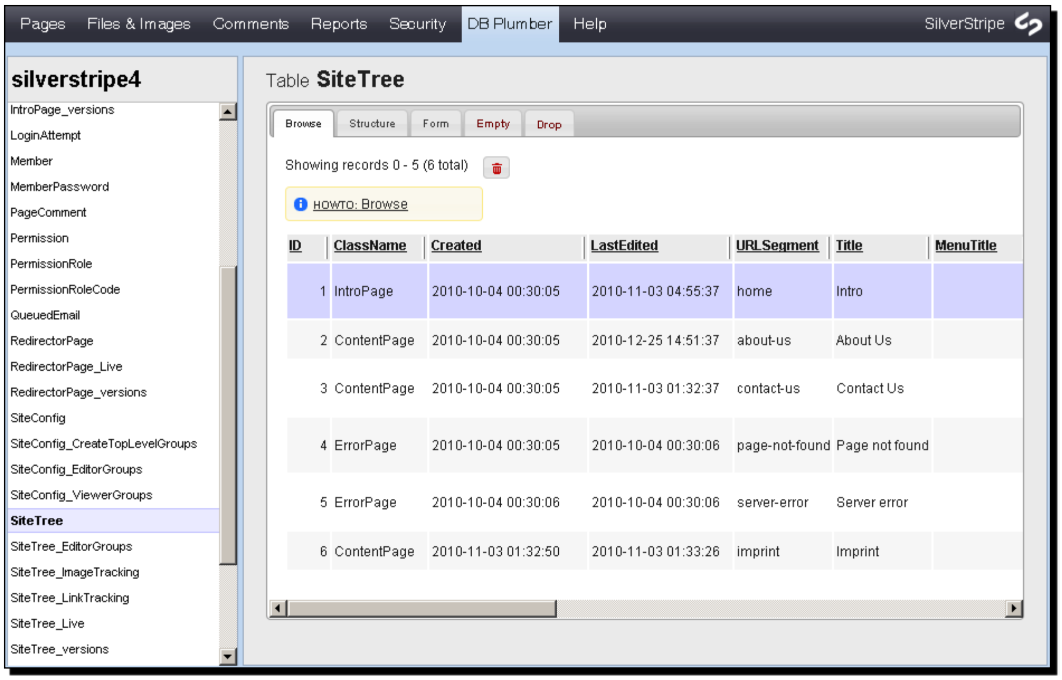
How does the code map to the database?

A database table maps to one base class. Each database row represents one instance of this class. Every database column stores one attribute of the class. So every class attribute is saved in one column of the instance's database row.

In order to illustrate this, take a look at the following code (we'll cover the details a little later):

```
"URLSegment" => "Varchar(255)",
"Title" => "Varchar(255)",
"MenuTitle" => "Varchar(100)",
"Content" => "HTMLText",
"MetaTitle" => "Varchar(255)",
```

And compare it to the specific table:



silverstripe4

IntroPage_versions
LoginAttempt
Member
MemberPassword
PageComment
Permission
PermissionRole
PermissionRoleCode
QueuedEmail
RedirectorPage
RedirectorPage_Live
RedirectorPage_versions
SiteConfig
SiteConfig_CreateTopLevelGroups
SiteConfig_EditorGroups
SiteConfig_ViewerGroups
SiteTree
SiteTree_EditorGroups
SiteTree_ImageTracking
SiteTree_LinkTracking
SiteTree_Live
SiteTree_versions

Table SiteTree

Showing records 0 - 5 (6 total)

ID	ClassName	Created	LastEdited	URLSegment	Title	MenuTitle
1	IntroPage	2010-10-04 00:30:05	2010-11-03 04:55:37	home	Intro	
2	ContentPage	2010-10-04 00:30:05	2010-12-25 14:51:37	about-us	About Us	
3	ContentPage	2010-10-04 00:30:05	2010-11-03 01:32:37	contact-us	Contact Us	
4	ErrorPage	2010-10-04 00:30:05	2010-10-04 00:30:06	page-not-found	Page not found	
5	ErrorPage	2010-10-04 00:30:06	2010-10-04 00:30:06	server-error	Server error	
6	ContentPage	2010-11-03 01:32:50	2010-11-03 01:33:26	imprint	Imprint	

The other fields are page-specific, but you should already know some of them. For example, **MetaTitle**, looks suspiciously like `$MetaTitle` from the second chapter, which we've used in the HTML page's `<title>` tag. These two are indeed connected; the database field stores the text rendered through the template.

In general, by using placeholders you can access methods from the Controller and both methods and properties from the Model.

Adding custom fields to a page

So you can easily fetch any of the **SiteTree** custom fields on a page. But how can we add our own properties?

Time for action – putting content into the database

In `themes/bar/templates/Includes/BasicInfo.ss` we've defined our site's basic information. Now assume that the opening hours of the bar change—something that might happen every now and then. Content editors should really be able to change that in the CMS without editing the template.

Therefore, in the template replace:

```
<div id="details-second">
  <div class="left">Opening hours:</div>
  <div class="right"><p>
    <b>Mon - Thu 2pm to 2am</b><br/>
    <b>Fri - Sat 2pm to 4am</b>
  </p></div>
</div>
```

with:

```
<div id="details-second">
  <div class="left">Opening hours:</div>
  <div class="right">$OpeningHours</div>
</div>
```

So now we need to store the required information in `$OpeningHours`. Note that this placeholder must contain the whole paragraph, including HTML tags.

Try to limit your placeholder's HTML to structuring text, including bold, italic, heading, paragraph, and list tags. The overall page structure, like `<div>` tags, should be defined in the template, otherwise you are blurring the MVC roles.

Which of our three models should we choose for that task? **Page**, **ContentPage**, **IntroPage**, or should we simply copy the necessary code into two or even three of those? Thinking back, we know that we need this information on both the **ContentPage** and **IntroPage** as we want to display that information on every page. So let's put it in those two, right?

Actually, this isn't such a good idea. Rather than write the code in two places, there is a way to write it in a single one. If we add `$OpeningHours` to **Page**, our two other pages will inherit it.

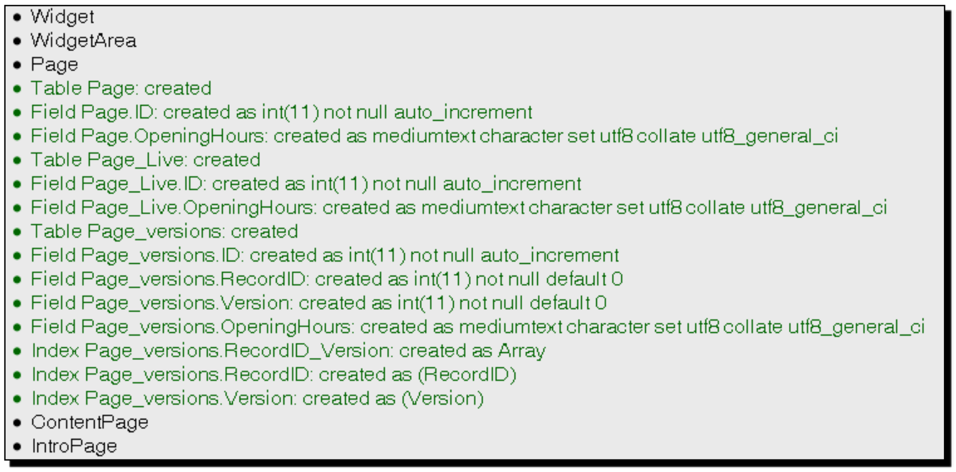
Temporary solution

Please note that this is only a temporary solution as there's an even better way of doing it. But let's do this step by step, and what we do now will aid your understanding later on.

So, replace the **Page** class with this code (leave the Controller as it is):

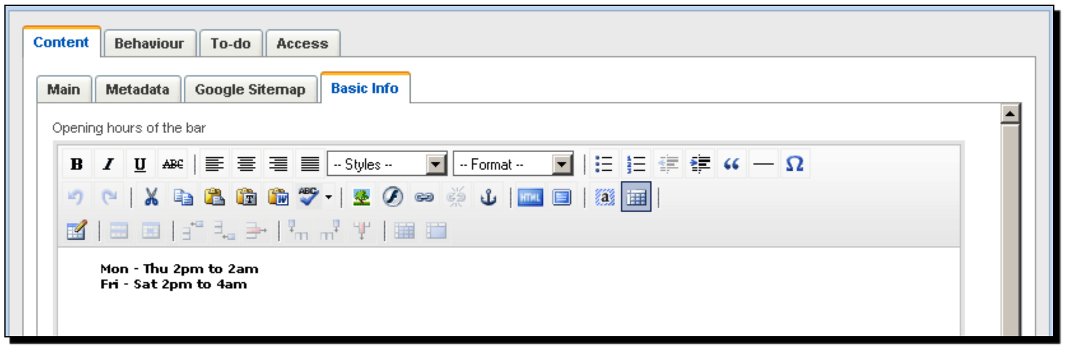
```
class Page extends SiteTree {

  public static $db = array(
    'OpeningHours' => 'HTMLText',
  );

  public function getCMSFields(){
    $fields = parent::getCMSFields();
    $fields->addFieldToTab(
      'Root.Content.BasicInfo',
      new HTMLEditorField(
        'OpeningHours',
        'Opening hours of the bar'
      )
    );
    return $fields;
  }
}
```

Call `/dev/build?flush=all`, which should produce the following output—green text highlights newly-added tables and fields:



```
• Widget
• WidgetArea
• Page
• Table Page: created
• Field Page.ID: created as int(11) not null auto_increment
• Field Page.OpeningHours: created as mediumtext character set utf8 collate utf8_general_ci
• Table Page_Live: created
• Field Page_Live.ID: created as int(11) not null auto_increment
• Field Page_Live.OpeningHours: created as mediumtext character set utf8 collate utf8_general_ci
• Table Page_versions: created
• Field Page_versions.ID: created as int(11) not null auto_increment
• Field Page_versions.RecordID: created as int(11) not null default 0
• Field Page_versions.Version: created as int(11) not null default 0
• Field Page_versions.OpeningHours: created as mediumtext character set utf8 collate utf8_general_ci
• Index Page_versions.RecordID_Version: created as Array
• Index Page_versions.RecordID: created as (RecordID)
• Index Page_versions.Version: created as (Version)
• ContentPage
• IntroPage
```

Refresh the CMS backend. Now you should see a new tab **Basic Info** on each page. Enter the following information into the CMS and apply the bold font on it:



Opening hours of the bar

Mon - Thu 2pm to 2am
Fri - Sat 2pm to 4am

Save and Publish the page. On the page where we just entered the information it will be displayed correctly. On pages where we've not yet added any content to this field it is simply empty in the frontend.

What just happened?

First off, we know why we've added the custom field in the **Page** class—thanks to inheritance.

The variable `public static $db = array();` has a special function, it is used for defining database fields. In our case we just defined one field called `OpeningHours` of the type `HTMLText`. We'll come back to the various types a little later, for now it's sufficient to know the obvious: this field stores our HTML.

Before we added the custom field, our custom classes were stored in the **SiteTree** table that we examined earlier. They had exactly the same fields as the base class, so there was no need for a separate table.

Now that we've added a custom field to **Page**, a dedicated table is required to store that information. It can't be added to **SiteTree** directly because that would also impact other subclasses like **VirtualPage**—that's not what we want.

Looking at the **Page** table in DBPlumber, we notice that there are just two fields, namely **ID** and **OpeningHours**. What about **ClassName**, **Created**, and **LastEdited**? These are valid for the base class. Subclasses only contain the ID of the parent record and the additional fields defined for them. You can also see that in the **ErrorPage** and **VirtualPage** tables.

[

What about those weird **Page_Live** and **Page_versions** tables? Those are created for any class or subclass of **SiteTree**. Unpublished but saved pages are stored in the **Page** table while published pages are saved in both this and the **Page_Live** table. **Page_versions** contains every saved version for later comparison or roll-backs.

]

Having created the necessary field in the database, how do we access it in the CMS? That's what the `getCMSFields()` method does. First we fetch all the fields already available (`parent::getCMSFields()`). Then we add our own field for the newly-created database entry and, finally, we return all the fields to the CMS for being displayed:

```
$fields->addFieldToTab(
  'Root.Content.BasicInfo',
  new HTMLEditorField(
    'OpeningHours',
    'Opening hours of the bar'
  )
);
```

Adding our own field is pretty dense, so let's break it down:

- We add a new tab called `BasicInfo` (or whatever you want to name it) to the already existing `Content` tab. Note that spaces are added before uppercase letters automatically, so in the CMS we see **Basic Info**. If we needed to add the new field to the first tab, it would have been `Root.Content.Main`.

- Then we create a new form field for HTML on the `OpeningHours` database field (the first argument). The second argument is an optional label; if we leave it out, the HTML field will be called **Opening Hours** after the database field name.
- Lastly, we can optionally add a third argument to the `addFieldToTab()` method, defining which field should follow after this one. As we're creating a new and currently empty tab, there is no need to use this option here. If you want to place our field in the **Main** tab, you can use the following code to add it above the **Content** field:

```
$fields->addFieldToTab(
  'Root.Content.BasicInfo',
  new HTMLEditorField(
    'OpeningHours',
    'Opening hours of the bar',
    'Content'
  )
);
```

Congratulations, you've just successfully added your first custom field to the CMS!

More data types and their form fields

Now that we've seen how creating our own database entries work and how to make them accessible in the CMS, it's time to add more data types.

Data type	Example	Form field
Boolean: Boolean field (true or false)	`'Understandable' => 'Boolean'`	`CheckboxField()`
Currency: Number with 2 decimal points	`'Price' => 'Currency'`	`CurrencyField()`
Date: Date field	`'StartDate' => 'Date'`	`DateField()`
Decimal / Double / Float: Decimal number	`'LitersOfVodka' => 'Decimal'`	`NumericField()`
Enum: List of strings, optional default value (here `unknown`)	`'Sex' => "Enum('male, female, unknown', 'unknown')"`	`DropdownField()`
HTMLText: HTML string, up to 2 MB	`'HTMLDescription' => 'HTMLText'`	`HtmlEditorField()`
HTMLVarchar: HTML string, up to 255 characters (here 64)	`'HTMLAbstract' => 'HTMLVarchar(64)'`	`HtmlEditorField()`
Int: Integer field	`'NumberOfPeople' => 'Int'`	`NumericField()`

Data type	Example	Form field
Percentage: Decimal number between 0 and 1	`'Evaluation' => 'Percentage'`	`NumericField()`
SS_Datetime: Field for both date and time	`'Start' => 'SS_Datetime'`	`DatetimeField()`
Text: String, up to 2 MB	`'Description' => 'Text'`	`TextareaField()`
Time: Time field	`'StartTime' => 'Time'`	`TimeField()`
Varchar: String, up to 255 characters (here 64)	`'Abstract' => 'Varchar(64)'`	`TextField()`

Data type and form field are not hard-wired, these combinations are just sensible examples. You can of course define an `'Email' => 'Varchar'` and then use a form field of `EmailField()`. But let's leave it at that for the moment; we'll gradually introduce the finer details as required.

Have a go hero – transfer the other BasicInfo.ss elements to the CMS

Now that you know all the basic elements it's time to get started on your own. Add the other elements of `BasicInfo.ss`, namely phone, contact, and address to the CMS.

"But wait! That creates a lot of redundant work for the content editors. Every time anything changes they'll have to make a change on every single page. There must be a better solution." You're absolutely right and that's exactly what we'll do next.

However, before we dive into that, what about our previous changes? Doing something, realizing it's not the ideal solution, and then undoing it, is a common process. So let's see how to handle that in SilverStripe:

1. Undo the changes in `Page.php` and `BasicInfo.ss`. Specifically, remove the `$db` variable and the `getCMSFields()` function as well as `$OpeningHours` in the template.
2. Enter `/dev/build?flush=all` again. This might produce an error; simply reload the page, and it should go away.
3. In the output or in DBPlumber you can see that our three newly-created tables (Page, Page_Live, and Page_versions) have been renamed. All of them now start with `_obsolete_`. SilverStripe will always do that for tables that are no longer related to the code. This should keep your installation clean without you losing any data.

4. If you have any relevant information in these tables, you can back them up with DBPlumber. We don't, so we can simply skip this step.
5. If you're sure that you don't need this information any more, you can delete the tables with DBPlumber and you have thereby removed every bit of our first experiment.

Removing a module works just the same way: remove the files, synchronize the database, and the relevant tables are automatically renamed. You can then backup and/or remove them.

Global custom fields

Do you remember `$SiteConfig.Title` from the second chapter? The **SiteConfig** object is available on all pages and there is just a single instance of it. Perfect, so how do we make use of this feature?

Configuration

It's a little more complicated than building upon the `Page` class, as there is no class we can directly extend. The reason is that the CMS only provides a single **SiteConfig** page, and rather than adding a new one, we want to extend the existing one. This technique is called Mixin or decorating and can be very handy in SilverStripe.

First, we need to register our own extension in `mysite/_config.php`:

```
DataObject::add_extension('SiteConfig', 'CustomSiteConfig');
```

Code

We've added an extension for `SiteConfig`, called `CustomSiteConfig`. So SilverStripe will now try to locate and load `mysite/code/CustomSiteConfig.php`. Time to create this file then:

```
class CustomSiteConfig extends DataObjectDecorator {

  public function extraStatics(){
    return array(
      'db' => array(
        'Phone' => 'Varchar(64)',
        'Address' => 'Varchar(64)',
        'Email' => 'Varchar(64)',
        'OpeningHours' => 'HTMLText',
      )
    );
  }

  public function updateCMSFields(FieldSet &$fields){
    $fields->addFieldToTab(
      'Root.Main',
      new TextField('Phone', 'Phone number')
    );
    $fields->addFieldToTab('Root.Main', new TextField('Address'));
    $fields->addFieldToTab(
      'Root.Main',
      new TextField('Email', 'Email contact address')
    );
    $fields->addFieldToTab(
      'Root.Main',
      new HTMLEditorField('OpeningHours')
    );
  }
}
```

Database fields

Let's start with the database's field definition. In a `SiteTree` subclass we can simply use the variable `$db`. Here we need to wrap it into an array inside the function `extraStatics()`. But otherwise it's the same as before. Besides the already used `OpeningHours` we add:

- A variable length string of up to 64 characters for the phone number. We don't use `Int` as we might need spaces and a leading +.
- For `Address` we use the same as it should be a good fit.
- For the `Email` field it's very tempting to use `new EmailField()` instead of its superclass `new TextField()`. This adds a validator for checking the e-mail address and you absolutely should use this field if possible. In our case, however, it's not possible as our spam-protected e-mail address (`contact at bar dot com`) isn't valid before being fixed by our JavaScript snippet.

Accessing properties in the CMS

For accessing the fields in the CMS we don't use `getCMSFields()` as before, but `updateCMSFields()`. The only difference is that the function call includes a reference to the already existing fields and we can simply add our attributes like before. As we're working with a reference, we don't need to return anything.

Call by value vs call by reference

There are two ways of passing variables to a method: by value or by reference. `function foo($bar)` is a call by value, the `$bar` variable is copied. `$bar` in the calling class is copied from, but not linked to `$bar` in the `foo($bar)` method.

`function foo(&$bar)` is a call by reference. Any change in the `foo(&$bar)` method automatically applies to the variable in the calling class as the original element is being referenced, rather than a copy.

Referenced variables include an ampersand in the method's definition.

Template

Changing the template is now easy. Replace the strings with the appropriate **SiteConfig** placeholder. So for `01 23456789` use `$SiteConfig.Phone` and so on. You can reuse the markup of the previous example, only add `SiteConfig` to it.

```
<a href="home">
  <img src="$ThemeDir/images/logo.png" alt="Logo" id="logo"/>
</a>
<ul id="details-first">
  <li>Phone: <b>$SiteConfig.Phone</b></li>
  <li>Contact: <a href="contact">
    <span class="mail">$SiteConfig.Email</span>
  </a></li>
  <li>Address: <a href="location">$SiteConfig.Address</a></li>
</ul>
<div id="details-second">
  <div class="left">Opening hours:</div>
  <div class="right">$SiteConfig.OpeningHours</div>
</div>
<a href="http://www.facebook.com/pages/">
  <img src="$ThemeDir/images/facebook.png" alt="Facebook"
    id="facebook"/>
</a>
```

Synchronize the database and you're ready to go

Synchronize your code with the database by calling `/dev/build?flush=all`. You won't get a new table, but your custom fields are added to the already existing **SiteConfig** table. Take a look at it and say hello to our old friends **Title**, **Tagline**, and **Theme**.

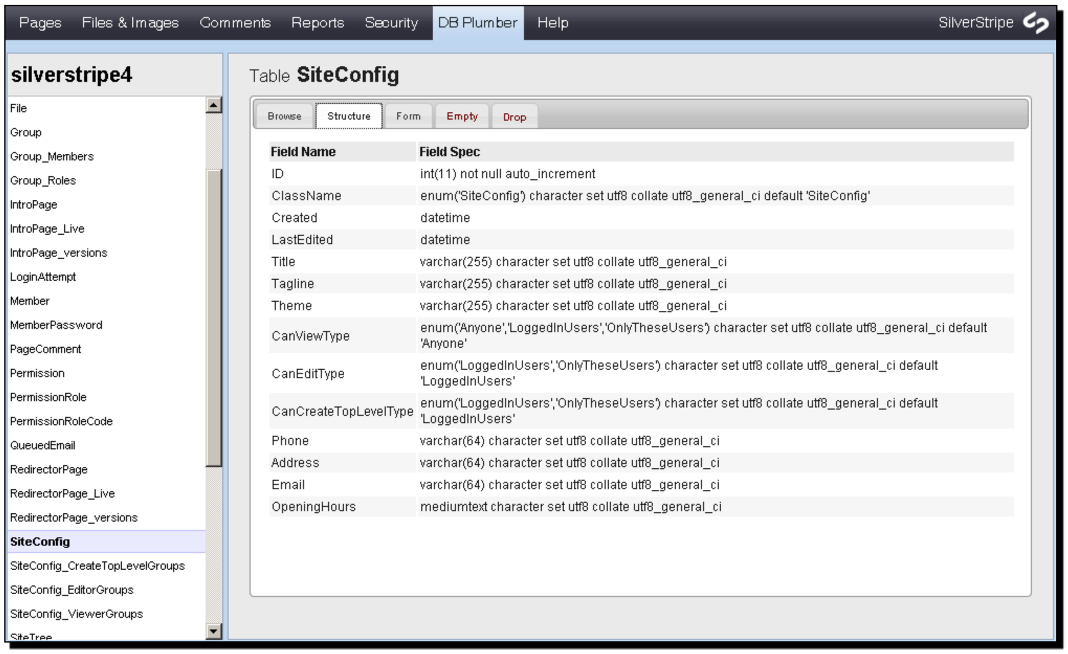
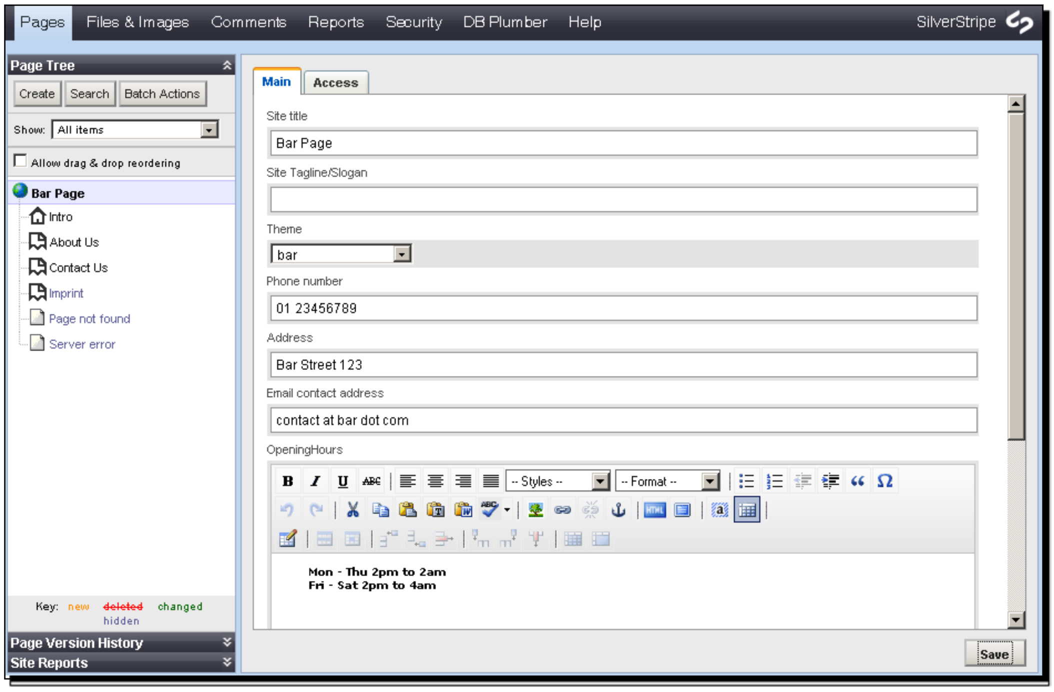


Table **SiteConfig**

Field Name	Field Spec
ID	int(11) not null auto_increment
ClassName	enum('SiteConfig') character set utf8 collate utf8_general_ci default 'SiteConfig'
Created	datetime
LastEdited	datetime
Title	varchar(255) character set utf8 collate utf8_general_ci
Tagline	varchar(255) character set utf8 collate utf8_general_ci
Theme	varchar(255) character set utf8 collate utf8_general_ci
CanViewType	enum('Anyone','LoggedInUsers','OnlyTheseUsers') character set utf8 collate utf8_general_ci default 'Anyone'
CanEditType	enum('LoggedInUsers','OnlyTheseUsers') character set utf8 collate utf8_general_ci default 'LoggedInUsers'
CanCreateTopLevelType	enum('LoggedInUsers','OnlyTheseUsers') character set utf8 collate utf8_general_ci default 'LoggedInUsers'
Phone	varchar(64) character set utf8 collate utf8_general_ci
Address	varchar(64) character set utf8 collate utf8_general_ci
Email	varchar(64) character set utf8 collate utf8_general_ci
OpeningHours	mediumtext character set utf8 collate utf8_general_ci

Now reload the CMS; go to the **SiteConfig** page and enter the information. **Save** the information and you can then access it on the frontend just like before—with the little difference that it can now be changed sitewide, from a single place in the CMS. Sweet!



Site title: Bar Page
Site Tagline/Slogan:
Theme: bar
Phone number: 01 23456789
Address: Bar Street 123
Email contact address: contact at bar dot com
OpeningHours:
Mon - Thu 2pm to 2am
Fri - Sat 2pm to 4am

How database values are fetched

Now that we've created our own placeholders, it's time to take a look at how values are actually fetched from the database. Basically, there are two approaches: pushing and pulling.

Pushing

The nice thing about this approach is that it's pretty easy and straight forward. That's the reason why most currently available systems employ this mode. However, it's not very efficient, as many values will never be used.

- Once a URL is requested, the underlying system determines which page needs to be rendered.
- Next, all the required data is fetched from the database and put into a very large array, which might look something like this:

```
array(
  'ID' => 1,
  'URL' => 'home',
  'MetaTitle' => 'Bar :: Intro',
  ...
  'SiteConfig' => array(
    'Phone' => '01 23456789',
    ...
```

- Finally, the placeholders in the template are replaced by the values in the relevant array. Now it should be clear why this approach is called pushing: data isn't specifically requested, rather everything is forwarded.

Pulling

SilverStripe uses the inverted approach: data is only processed when needed. Additionally, instead of using an array, PHP objects are used. As you've already seen, SilverStripe focuses on object-orientation so this is a very good match.

- The first step is the same as in the pushing approach: once a URL is requested, the page which needs to be rendered is determined.
- SilverStripe creates a page object and populates it with raw data from the database. While the data is fetched in one go (doing many small queries would be very inefficient), it's not transformed to PHP automatically.
- The page object automatically provides methods to access the raw data. For example, `$MetaTitle` in the template would call the method `MetaTitle()`. This grabs the data (having been cached or freshly fetched from the database), processes it, and adds it to the template.

This approach saves a lot of unnecessary processing and memory use: calculating and keeping in memory many never-used values such as `$Date.Ago`, `$Data.Day`, and `$Date.DayOfMonth`, is not very efficient.

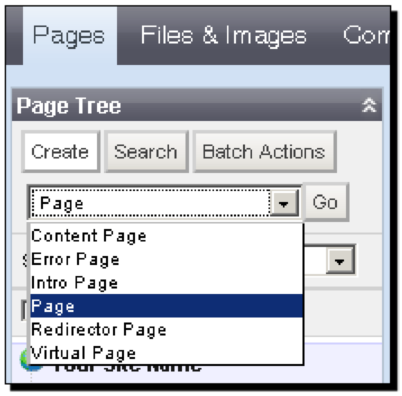
Additionally, related objects cannot only be accessed in the Controller, but also in the template. We'll come to that later.

Taking the Model further

So we can add more fields to the CMS, great. But sometimes less is more. Let's see if we can do a little clean-up, making the life of our content editors as well as our own easier by providing less "opportunities" for doing something wrong, which will hopefully result in fewer support calls.

Removing unused Page types

At the moment we can create the following pages in the CMS:

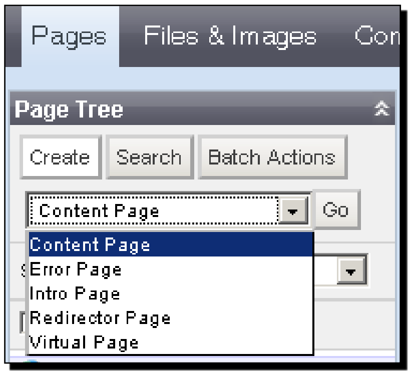


However, **Page** isn't used at the moment, it's only needed for inheritance at the code level. In order to avoid any confusion, we'd like to hide the page in the CMS. Fortunately, SilverStripe has a mechanism for solving this problem. Add a single line to the Model of either `ContentPage` or `IntroPage` and you're done:

```
public static $hide_ancestor = 'Page';
```

This is just an example for this specific setup. Often the base `Page` type is used for most pages. But it can also be handy to have a base page available, which is only used for inheritance.

After reloading the CMS (no database rebuild required), it will look like this:



Showing the Page type in the Page Tree

While we're at the page tree, it's a little unnerving not knowing what page type is used for which page. Having to click on each page and navigating to the right tab is not a perfect solution.

Adding an image

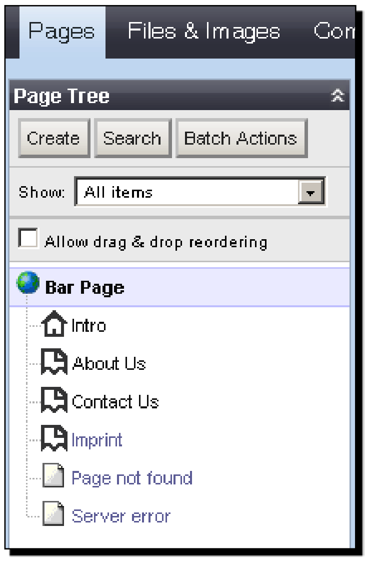
SilverStripe provides an easy solution, which displays an image of our choice for different page types rather than the generic default one:

```
public static $icon = 'mysite/icons/intro';
```

This sets up the image `mysite/icons/intro-file.gif` as the icon for this page. You must, of course, add the directory and file before it will work. Note that the `-file.gif` is automagically appended—make sure to name your files correctly.

Why don't we put the icons into the theme's folder? You're free to do so. We didn't because we're assuming it's a backend issue not associated with the current theme. We'll reuse this icon set for all of our pages so it's easier to associate the right page type with the icon.

After adding an icon for the intro and content class, our page tree might look something like this:



Don't change the file ending

What should you do, if you don't want to call your image `-file.gif`? Trust us, you should. Convention over configuration is at work for you again. Take a look at the page's source code in your browser. You'll find something like this:

```
_TREE_ICONS['BrokenLink'] = {
  fileIcon: 'cms/images/treeicons/brokenlink-file.gif',
  openFolderIcon: 'cms/images/treeicons/brokenlink-openfolder.gif',
  closedFolderIcon: 'cms/images/treeicons/brokenlink-closedfolder.gif'
};
```

You'll probably have already guessed what this does:

- If a page has no subpages, `-file.gif` is used, for example, `intro-file.gif`.
- If a page has subpages, you can add specific icons for open and closed "folders". It's actually only a page, but as it contains subpages it's called a folder.
- You can provide an icon with `-openfolder.gif` and `-closedfolder.gif` appended, for example, `intro-openfolder.gif` and `intro-closedfolder.gif`.
- If such a folder icon doesn't exist, it falls back to the "file" variant.

So you simply need to define the base name, add the images named accordingly and everything else is done for you automagically.

Using constants

There is one minor enhancement we can still make. So far we've carefully avoided hard-coding any paths in our system. Let's do the same for our icons.

First add the following code to `mysite/_config.php`:

```
global $project;
$project = 'mysite';
define('PROJECT_DIR', $project);
```

Define a constant for our icon in the `mysite/code/IntroPage.php` file:

```
define('INTRO_ICON_PATH', PROJECT_DIR . '/icons/intro');
```

Be sure to place this outside of the `IntroPage` class as we're creating a so-called class constant. It's subsequently available in both the Model and the Controller, so now we can use it in the `IntroPage`:

```
public static $icon = INTRO_ICON_PATH;
```

We could have simply called it `ICON_PATH` but that would later pose a problem when using inheritance—on which, more soon. Therefore, we'll simply create a unique instance.

More SiteTree magic

There are also some more things you can do in a similar fashion.

For example, you can define whether a page can be root or only a child page in the page tree:

```
public static $can_be_root = false;
```

You can also specify what child pages a page can have, for example, only `ContentPage`:

```
public static $allowed_children = array('ContentPage');
```

Assume that your website has a blog. By enabling content editors to only add blog posts within a blog section, you improve the integrity of your site and make it easier for authors to do their work without error.

Those are just two examples, there are many more things you can influence or set up, but we'll leave it at that.

Cleaning up content fields

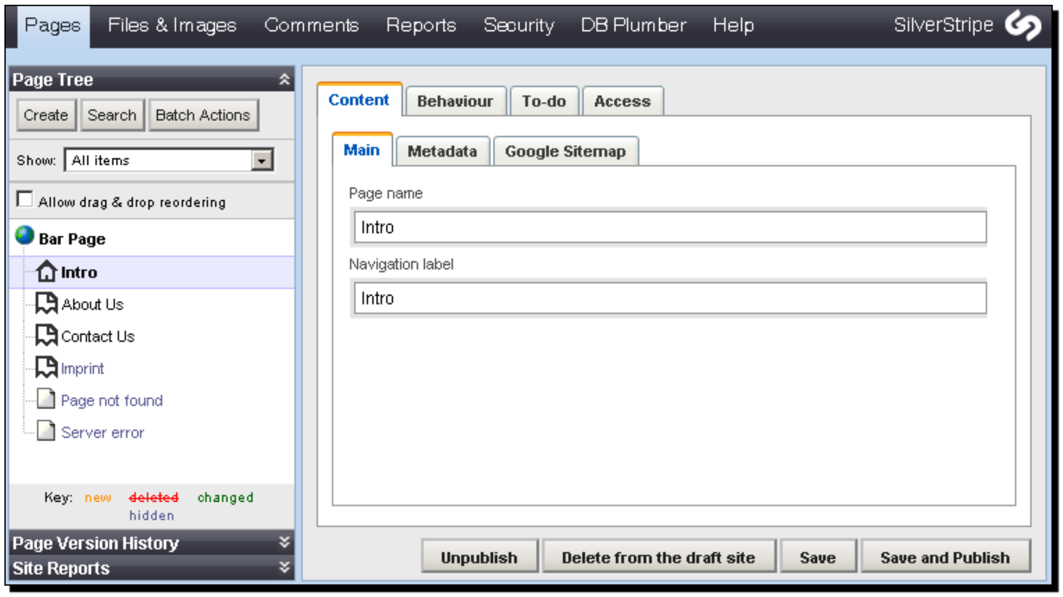
Now that the page tree is all cleaned up, it's time to turn our attention to the content fields. There are some fields which are not used, so rather than distract content editors, let's remove them.

On our intro page, we don't need the content area. It's never used in the template, so remove it by adding the following code to the Model:

```
function getCMSFields(){
  $fields = parent::getCMSFields();
  $fields->removeFieldFromTab('Root.Content.Main', 'Content');
  return $fields;
}
```

We already know the `getCMSFields()` function. What's new is `removeFieldFromTab()`, which removes the **Content** field from the **Main** tab inside **Content**, which is inside **Root**. **Root** is the overall container and the names of the other two are displayed in the CMS. It's just the reverse of `addFieldToTab()`.

If you reload the CMS, the intro page will now look like this—much cleaner:



Page name: Intro

Navigation label: Intro

Our `CustomSiteConfig.php` is also a bit crowded. Let's remove the unused **Title** and **Tagline** fields:

```
public function updateCMSFields(FieldSet &$fields){
  $fields->addFieldToTab('Root.Main',
              new TextField('Phone', 'Phone number')
  );
  $fields->addFieldToTab('Root.Main', new TextField('Address'));
  $fields->addFieldToTab('Root.Main',
              new TextField('Email', 'Email contact address')
  );
  $fields->addFieldToTab('Root.Main',
              new HTMLEditorField('OpeningHours')
  );
  $fields->removeFieldFromTab('Root.Main', 'Title');
  $fields->removeFieldFromTab('Root.Main', 'Tagline');
}
```

There's nothing really new here, just add the highlighted lines and you're good to go.

Setting a meta title prefix

To finish this section off, let's briefly take a look at a handy search engine enhancement.

Assume you want to prefix all of the pages' meta title with `"Bar :: "` and then append the default title. That's what this snippet does, add it to the Model class of `mysite/code/Page.php`:

```
public function getMetaTitle(){
  return "Bar :: " . $this->Title;
}
```

Reload the CMS and you'll see the difference on the **Metadata** tab.

Managing relations

Besides having fields, database tables can have relationships with other tables (within the same or another database). That's why they are called relational databases.

Time for action – linking pages together

One such example would be to amend our intro page, by allowing it to link to another page. So let's build that:

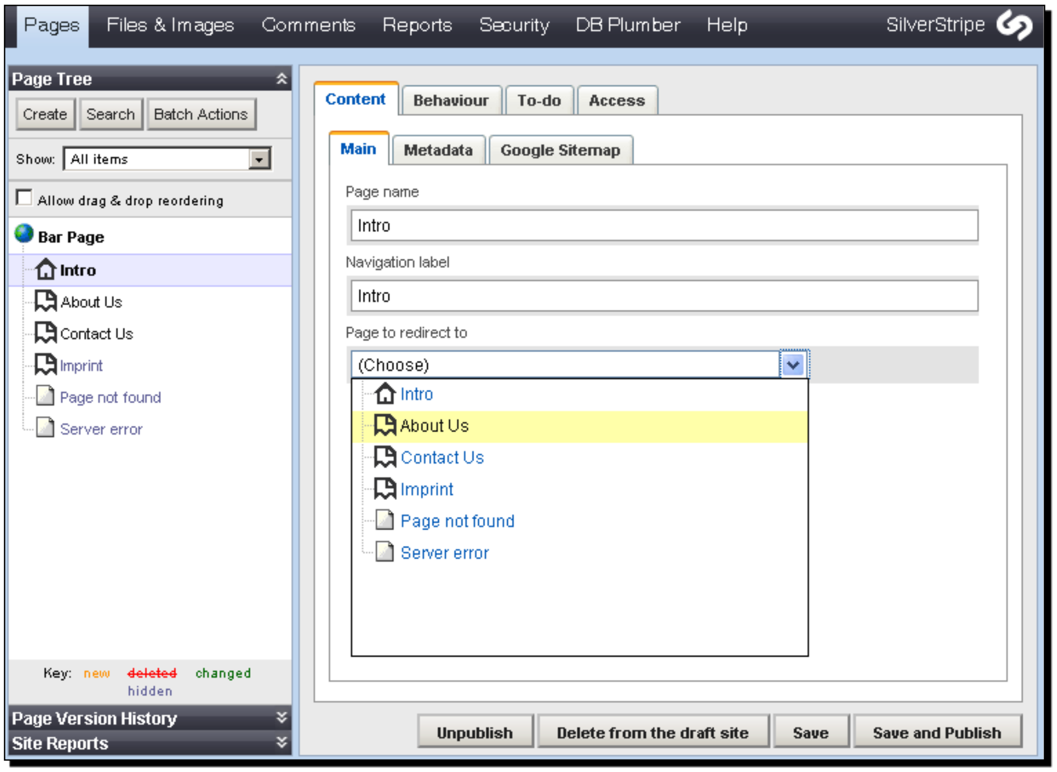
1. Inside the Model of `mysite/code/IntroPage.php` add:

   ```
   public static $has_one = array(
     'PageRedirect' => 'SiteTree',
   );
   ```

2. In the same file, change the method `getCMSFields()` to:

   ```
   public function getCMSFields(){
     $fields = parent::getCMSFields();
      $fields->addFieldToTab(
        'Root.Content.Main',
        new TreeDropdownField(
          'PageRedirectID',
          'Page to redirect to',
          'SiteTree'
        )
      );
     $fields->removeFieldFromTab('Root.Content.Main', 'Content');
     return $fields;
   }
   ```

3. Rebuild the database.
4. Reload the CMS and go to the intro page.
5. You should now be able to select the page to redirect to.



Page name: Intro

Navigation label: Intro

Page to redirect to: (Choose)

- Intro
- About Us
- Contact Us
- Imprint
- Page not found
- Server error

6. In `themes/bar/templates/Layout/IntroPage.ss` replace:

```
<a href="start">
```

with:

```
<a href="$PageRedirect.Link">
```

7. Reload the intro page (don't forget flushing) and upon clicking on the background it should redirect you to the page that you have selected in the CMS.

What just happened?

Just a few lines of code have been added but there is quite a lot of power behind them, so let's take a closer look at it step-by-step.

Definition

Instead of using `$db` to define the database fields, we define a relationship using `$has_one`. As the name implies, this ties the current object to one other object.

In our case it's another page so we're referencing the base class of all pages: `SiteTree`. We're calling this relationship `PageRedirect`, just like with `$db`.

[Don't be tempted to simply call it `Redirect`. This name is already used in the template and will lead to a collision, breaking the example.]

Adding relationships to the CMS

After the definition we're making it available in the CMS. The interesting part of it is:

```
new TreeDropdownField(
  'PageRedirectID',
  'Page to redirect to',
  'SiteTree'
)
```

The function provides a drop-down field, based on a tree data structure—as the name implies. Let's take a look at the three arguments supplied:

- The relationship is called `PageRedirect`, we reference it with `PageRedirectID` (don't overlook the attached `ID`). Leave the object-oriented world for a moment and think about how the database stores a relationship: it stores the (unique) ID of the related row/object, being called a foreign key. That's the reason why we don't reference the other object itself but rather its ID.
- Next a label is provided; you can provide any helpful text here.
- Finally, the available values are provided for the function. `SiteTree` is the object containing all pages (as they are a subclass of it), so this gives us what we need.

So we're having a drop-down with all of the pages currently available in the system. After selecting one, its ID is saved for referencing the correct page in the template.

Using it in the template

In the template we replace our hard-coded link with the right value from the database:

```
$PageRedirect.Link
```

Note that the following code produces the same result. That's how flexible the template is:

```
<% control PageRedirect %>
  $Link
<% end_control %>
```

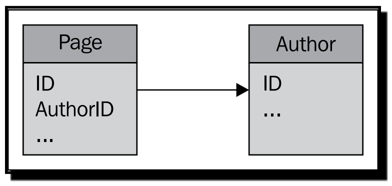
We're accessing the referenced object (not its ID any more) and fetch its URL. That's the correct link we were after. There's nothing you need to do for that, SilverStripe automagically provides this functionality for relationships.

Once you understand the basic principles, it's not so hard any more, so keep trying.

Complex relationships

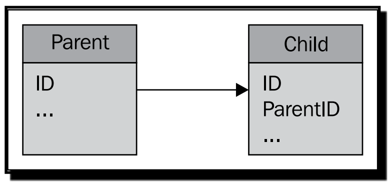
`$has_one` is the easiest relationship, also called 1:1 as it relates one object to another. On a database level, the (foreign) key of one object is stored in an extra column in the other object. It's saved only on one side of the relationship and is also only accessible from this side.

As an example, imagine that a **Page** has exactly one **Author**. This would enable readers to get more details about the author of a specific page.



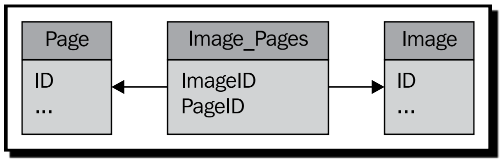
If you need to navigate in both directions, you'll also need a `$has_many` relationship called 1:n. This provides the other direction (if you need it).

For example, assume there is one **Parent** page, which has many **Child** pages and each child has exactly one parent. This will be useful in a page's menu, when you want to navigate into both directions. We'll come back to that later.



Finally, there is also the `$many_many` relation, also called m:n. It allows you to link any number of objects from one class to any number of objects in another class.

Here we're having many pages and many images. One **Page** can embed many images, whereas one **Image** can be embedded on many pages. For this concept a so-called join table (the one connecting the two classes) is necessary. More details a little later.



Queries

For the sake of completeness we'll also take a quick look at queries, or how to access objects. We'll get back to these with some detailed examples later on:

- `DataObject::get_by_id($class, $id)`: Gets the entry with the `$id` of the given class, for example: `DataObject::get_by_id('SiteTree', 1);`.

  This would fetch the first row of the `SiteTree` table from the database—if it hasn't been deleted. Watch out for empty results.
- `DataObject::get_one($class, $filter)`: Fetches at most one entry of the given class, satisfying the provided filter argument, for example: `DataObject::get_one('SiteTree', "URLSegment = 'home'");`.

  This will fetch the page with the URL of "home", basically your index page, if it exists. Watch out for empty results.
- `DataObject::get($class, $filter, $sort, $join, $limit)`: This is like a fully featured `SELECT` statement and you'll easily guess what each parameter is good for. Note that only the first argument is required and, as always, watch out for empty result sets.

By using these features of the ORM, you'll hardly need plain old SQL. As it's easier and more portable across databases, you should use the ORM whenever possible. However, it's still possible to use your own, raw SQL queries, in case you're trying to do something very complex or you know a more efficient way than the ORM for a specific query. If you are sure you need raw SQL, look into the class `SQLQuery`.

So far we've totally ignored how database entries are written to the database outside the CMS. We'll come back to that, but at the moment our input options are limited to the CMS so there's no point in going beyond that right now.

Security

In order to wrap up this chapter of storing and retrieving data, it's important to quickly address security. When accessing the database, always take steps to improve the security of your site:

- So called **SQL Injections** are a wide-spread disease in web applications. By not properly checking user-supplied inputs, they allow an attacker to write arbitrary data to the database, often compromising the website altogether.
- While SQL injections are mainly about input to your database, you should also take care of the output. **Cross-site scripting** (**XSS**) describes a form of vulnerability where an attacker can make an application output malicious content. This is often used to trick an unsuspecting visitor into something. An attacker might, for example, try to add a very prominent message to a website, stating that the domain name has changed and every visitor should now use `http://malicious.com` instead of the current page.

There are two things you should do whenever working with user-supplied inputs or outputs: cast and escape. Fortunately, SilverStripe supports these countermeasures very well.

Casting

PHP is a weakly typed language. Whenever you feed a variable into it, it will try to figure out the data type itself, unless you cast it in the code.

Where possible, cast your data type manually, especially when accessing the database. If you want to use `DataObject::get_by_id()` and you want to provide a dynamic ID, you need to make sure it's really an integer.

For example, `DataObject::get_by_id('SiteTree', (int)$id)` will cast `$id` to an integer. If `$id` already is a number, nothing changes. If `$id = "3;drop table SiteTree"`, `(int)$id` results in 3—as the full string isn't a valid number. If you're casting you'll get the first line of the following example, if not then the second one:

```
SELECT * FROM SiteTree WHERE ID = 3;
SELECT * FROM SiteTree WHERE ID = 3; DROP TABLE SiteTree;
```

Instead of (possibly) dropping your whole `SiteTree` table, you'll only try to fetch an ID, which might not exist. So when you cast manually, you'll avoid any trouble.

Actually, this won't work as `DataObject::get_by_id()` itself escapes the ID, but it's nevertheless a good idea to cast wherever possible.

The two most common data types you'll use for casting are `(int)` and `(boolean)`.

Additionally, we've already taken a look at template casting. For example, `$Value.XML` escapes any XML and HTML tags in the placeholder `$Value`. If the value had been user supplied, an attacker can only provide plain text instead of HTML or JavaScript.

Escaping

If you need to use a string when accessing the database, casting won't help you as a string can contain anything. So when using strings, escape them:

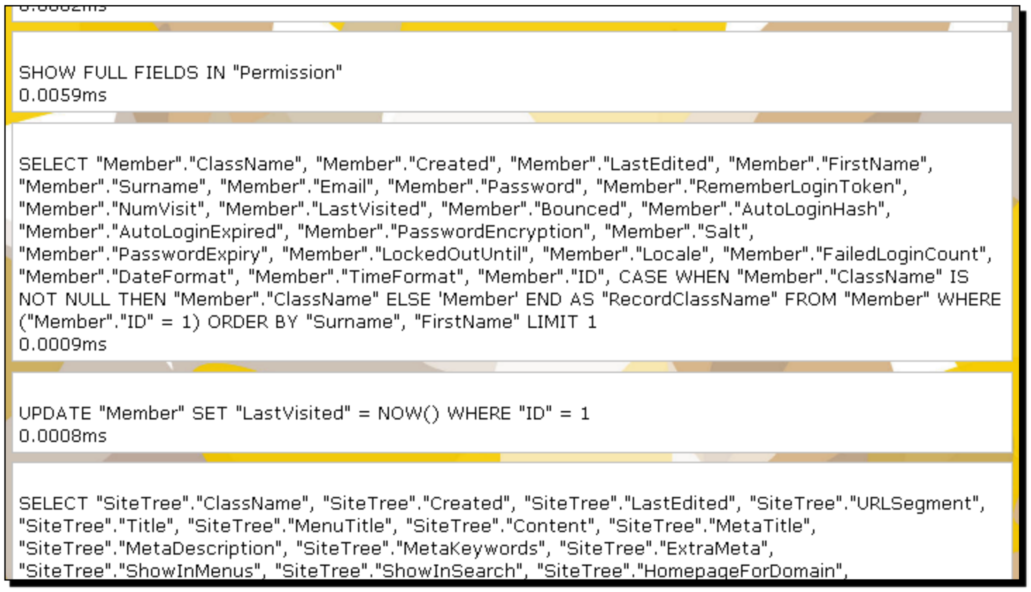
```
$good = Convert::raw2sql($_GET['evil']);
DataObject::get_one('SiteTree', "URLSegment = '$good'");
```

Here escaping is crucial as an attacker can insert arbitrary SQL statements via the GET parameter of `evil`. `Convert::raw2sql()` strips out anything harmful; use it as much as you can.

Debugging queries

There are two useful URL parameters you can use when debugging queries:

- `?showqueries=1`: Lists all queries executed on the current page.



```
SHOW FULL FIELDS IN "Permission"
0.0059ms

SELECT "Member"."ClassName", "Member"."Created", "Member"."LastEdited", "Member"."FirstName",
"Member"."Surname", "Member"."Email", "Member"."Password", "Member"."RememberLoginToken",
"Member"."NumVisit", "Member"."LastVisited", "Member"."Bounced", "Member"."AutoLoginHash",
"Member"."AutoLoginExpired", "Member"."PasswordEncryption", "Member"."Salt",
"Member"."PasswordExpiry", "Member"."LockedOutUntil", "Member"."Locale", "Member"."FailedLoginCount",
"Member"."DateFormat", "Member"."TimeFormat", "Member"."ID", CASE WHEN "Member"."ClassName" IS
NOT NULL THEN "Member"."ClassName" ELSE 'Member' END AS "RecordClassName" FROM "Member" WHERE
("Member"."ID" = 1) ORDER BY "Surname", "FirstName" LIMIT 1
0.0009ms

UPDATE "Member" SET "LastVisited" = NOW() WHERE "ID" = 1
0.0008ms

SELECT "SiteTree"."ClassName", "SiteTree"."Created", "SiteTree"."LastEdited", "SiteTree"."URLSegment",
"SiteTree"."Title", "SiteTree"."MenuTitle", "SiteTree"."Content", "SiteTree"."MetaTitle",
"SiteTree"."MetaDescription", "SiteTree"."MetaKeywords", "SiteTree"."ExtraMeta",
"SiteTree"."ShowInMenus", "SiteTree"."ShowInSearch", "SiteTree"."HomepageForDomain",
```

- `?previewwrite=1`: Lists all insert and update queries, but doesn't execute them. This is useful for simulating some action to see what it really does.

[Note that your site needs to be in development mode for this to work.]

Visible or invisible?

Now that we've covered a lot of features in the MVC roles, it's time to consider one specific setting: we've set the visibility of all functions and variables, but why did we actually do that? There are three possible values for the visibility:

- `public`: This element can be accessed from anywhere and is the default behavior if nothing is specified.
- `protected`: This element can only be accessed from within the same class or one of its subclasses.
- `private`: This element can only be accessed from within the same class.

Why do we care? Using visibility you can protect your code—from coding errors, inflicting unwanted side effects, as well as users accessing functionality that they shouldn't be able to use.

In the Model

You don't have much choice in the Model. The definition of the database, the static variables and the method adding and removing fields from the CMS must be public. If they are not, you'll get an error message.

You could leave out all the `public` statements, but it might be a good idea to keep them for better readability.

In the Controller

Things are more interesting in the Controller:

- Make methods that should be accessible via the URL `public`
- Make helper methods and the ones used in the templates `protected`
- `init()` must be `public`

Now it becomes obvious why we would care for the visibility of these method declarations in `mysite/code/Page.php`:

```
public function init(){
protected function TemplateMethod(){
protected function helperFunction(){
```

This is vital in shielding destructive or internal methods from URL access as we neither want to endanger our data nor disclose any sensitive information.

To further constrain which public functions should be available via the URL (currently we don't have any), you can use `$allowed_actions` in the Controller (not in the Model).

Assume a page is available under the address `/mypage`. With the following code you can access `/mypage/visible` but not `/mypage/invisible`:

```
public static $allowed_actions = array('visible');

public function visible(){ }

public function invisible(){ }
```

As we're not having any such methods in our code, we can simply add `public static $allowed_actions = array();` (not allowing any access) to `Page.php`, `ContentPage.php`, and `IntroPage.php`. It's not required at the moment, but a convention that you should follow. Better be safe than sorry.

Summary

We have now covered all parts of MVC. That's a solid base we can build on further.

We specifically took a look at DBPlumber and why it's useful. Next, we examined the basic database structure of SilverStripe, which field types are available, and how to add our own fields to the Model. Additionally, we learned how to display or hide database fields in the CMS. We took a look at the relationship model in SilverStripe, database queries in general, and some basic security techniques that should be utilized commonly in your applications.

In the examples, we extended the **SiteConfig** with our own fields and used them on our site. Furthermore, we used some of the `SiteTree` magic to spice up the CMS and make it more user friendly. And for developers we looked at quite a few debugging features within SilverStripe.

Now that we've covered the Model, let's throw in some more advanced topics. In the next chapter we'll take an in-depth look at SilverStripe's configuration.

5

Customizing Your Installation

Now that we've covered SilverStripe's general architecture, let's take a look at one other important aspect before implementing some more advanced features: Configuring our project site. What are the most important features, as well as common settings you should apply in your installation?

In this chapter we shall learn how to:

- Configure live and test environments, and specifically what their implications are
- Log and secure your site
- Customize the CMS in general and the rich text editor in specific
- Manage your own source code and learn how SilverStripe manages its code

So let's take control of the system...

Configuration in general

Each module in SilverStripe is located in a top-level folder and is configured through a `_config.php` file directly in this folder. The configuration file must have this name to be automatically picked up by the system. Note that this file is the only one based on procedural code in SilverStripe—everything else must be wrapped into classes.

In the configuration file you can call almost any static method as classes are loaded as needed.

We'll now focus on the configuration of our code, being defined in the file `mysite/_config.php` which is generated when you install SilverStripe. We'll leave out internationalization, module-specific configurations, and other parts, which we'll cover later on. The options discussed next are just some of the most important, general purpose SilverStripe settings.

The different configuration files

The general purpose configuration of a module is provided by its own configuration file, in its dedicated folder. However, you really shouldn't change code in core or third-party files (unless you have a really good reason for it), as this will make updates painful later on. Putting your custom configuration options into `mysite/_config.php` is a much better approach, especially when updating.

Default settings

The configuration `mysite/_config.php` looks something like this after the installation (comments stripped):

```
<?php

global $project;
$project = 'mysite';

global $databaseConfig;
$databaseConfig = array(
  "type" => 'MySQLDatabase',
  "server" => 'localhost',
  "username" => 'root',
  "password" => '',
  "database" => 'silverstripe5',
  "path" => '',
);
MySQLDatabase::set_connection_charset('utf8');

SSViewer::set_theme('blackcandy');

i18n::set_locale('en_US');

SiteTree::enable_nested_urls();
```

The folder of the page specific code is defined first. By default, this is `mysite`. We could now rename the `mysite` folder to `bar` and set `$project = 'bar';` but we'll just leave it as it is to spare us any future path confusion. The site-specific code is just another module.

Next, the database is configured. The default database is MySQL and this is the one we will use this book. PostgreSQL, SQLite, MS SQL Server, and Oracle are supported through dedicated modules. `server`, `username`, `password`, and `database` are obvious, while `path` is not used for MySQL. Finally the character encoding is set to UTF-8 so we won't run into issues with international characters such as German umlauts (äöü). We also use UTF-8 encoding in the template so everything should be displayed correctly and consistently.

`SSViewer::set_theme()` is used for setting the default theme. We could set it to `bar` here or override it in the CMS as we've already done.

For setting the localization use `i18n::set_locale()`. This setting also defines the value of `$ContentLocale`. If you were using German texts in your CMS you'd set this to `de_DE` instead of `en_US`.

Finally you're enabling nested URLs. If you delete or comment this statement out all pages are on the same level. Assume you have a subpage of `/home` in the CMS:

- If you have enabled nested URLs, the page's address would be `/home/subpage`
- If you have disabled nested URLs, the address would simply be `/subpage`

You'll need to decide if you want to use flat or hierarchical URLs. SilverStripe supports both, and you can change this anytime you want. Internal links will still work after changing it, but remember that links from external pages might break. So be careful on a live site.

Environment

We've already changed the environment mode in the previous chapters, but let's take a more detailed look at it now.

Probably the most important setting when developing a website is the environment mode. It's not explicitly set by the installer and defaults to `Director::set_environment_type('live')`. That's what you want in production, but not for development. To put your page into the development mode replace `live` with `dev`. Additionally there is also the `test` mode.

dev

Displays any (error) messages directly in the browser. Additionally you can call `/dev/build` without being logged in as an administrator—even without being logged in at all.

You can use `Director::isDev()` to check if you're currently in development mode. This can be handy to set up the database connection depending on the current mode. You will need to change the environment but can leave the rest of the code unchanged when deploying the application.

```
global $databaseConfig;
if(Director::isDev()){
  $databaseConfig = array(
    "type" => "MySQLDatabase",
    "server" => "localhost",
    "username" => "dev-user",
    "password" => "dev-password",
    "database" => "dev-database",
  );
} else {
  $databaseConfig = array(
    "type" => "MySQLDatabase",
    "server" => "localhost",
    "username" => "live-user",
    "password" => "live-password",
    "database" => "live-database",
  );
}
```

test

This mode behaves like `live` but should be used for testing or staging installations. You could use it to disallow anonymous access to your page, so that only developers and the client can see the current state of the application. `BasicAuth::protect_entire_site()` forces you to login before getting access to any page.

```
if(Director::isTest()){
  BasicAuth::protect_entire_site();
}
```

live

All error messages are replaced by generic ones (the `500` error message instead of the offending line). This looks more professional and you won't accidentally reveal sensitive information. Additionally you'll need to login as an administrator to flush the cache or rebuild the database.

Check for this mode with `Director::isLive()`.

Defining a development server

You can define that a site is run in development mode regardless of the current setting if you access it on a specific address. So if you'd access a page on the same system (`http://localhost`) it would be in development mode, from any other system it would follow the `Director::set_environment_type()` setting.

Forcing a page into development mode if it's accessed from the same server or under the URL `http://dev.mysite.com` would be done by this code segment:

```
Director::set_dev_servers(
  array(
    'localhost',
    '127.0.0.1',
    'dev.mysite.com',
  )
);
```

Server-wide configurations

When running multiple SilverStripe instances with similar configurations on the same server it's tiresome to set things up again and again. And don't forget we're striving for the DRY principle! Additionally, you'll want to run all sites in `dev` mode on your development machine but in `live` on the production system.

There's a pretty elegant solution for all this: you can define a system-wide base configuration. This configuration file must have the name `_ss_environment.php` and can be placed in your site's base folder, the parent of it or the grandparent.

For example if you have a folder `Webserver/` for all your projects, the system-wide configuration would be in `Webserver/_ss_environment.php`, and your projects would be in the folders `Webserver/silverstripe1/`, `Webserver/silverstripe2/`, and so on.

> Due to the different path structures in Windows on the one side and Linux, Mac, and BSD on the other, we'll just use these generalized paths to illustrate the examples. You shouldn't need to set absolute paths yourself so don't worry about compatibility. If you're working on Windows, simply think of it as `C:\Webserver\silverstripe1\`, otherwise as `/home/www/silverstripe1/` (or wherever your webroot is located). And always use forward slashes—Windows will use them correctly, while you'll have problems the other way around.

An example for `_ss_environment.php` on your development machine could look like this:

```
<?php

define('SS_ENVIRONMENT_TYPE', 'dev');

define('SS_DATABASE_SERVER', 'localhost');
define('SS_DATABASE_USERNAME', 'root');
define('SS_DATABASE_PASSWORD', '');
```

Your `mysite/_config.php` would then pick up these constants and could be limited to:

```
<?php

global $project;
$project = 'mysite';

global $database;
$database = '(databasename)';

require_once("conf/ConfigureFromEnv.php");
```

Obviously you'd have to replace `databasename` with your site's real database name. Database user, password, and server are generally the same on one system, the database itself is the only part that is different for each site.

However, this assumes that the database name is the same on your development and production system. If that's not the case, you could always throw in an additional `if(Director::isLive())`.

With the `require_once` method, you load the core file `sapphire/conf/ConfigureFromEnv.php` (only classes are automatically included) and not the system wide configuration. The configuration file is then loaded and set up by the core file.

You can only set constants in `_ss_environment.php`. For a complete list see the comments in the core file `sapphire/conf/ConfigureFromEnv.php`.

Logging

Take a look at the following code, what does it accomplish in `mysite/_config.php`?

```
SS_Log::add_writer(
  new SS_LogFileWriter(Director::baseFolder() . '/logs/ss.log'),
  SS_Log::ERR
);
```

```
SS_Log::add_writer(
  new SS_LogFileWriter(Director::baseFolder() . '/logs/ss.log'),
  SS_Log::WARN
);
SS_Log::add_writer(
  new SS_LogFileWriter(Director::baseFolder() . '/logs/ss.log'),
  SS_Log::NOTICE
);
ini_set("log_errors", "On");
ini_set("error_log", Director::baseFolder() . "/logs/php.log");

if(Director::isLive()){
  SS_Log::add_writer(
    new SS_LogEmailWriter('mail@mysite.com'),
    SS_Log::ERR
  );
  SS_Log::add_writer(
    new SS_LogEmailWriter('mail@mysite.com '),
    SS_Log::WARN
  );
  SS_Log::add_writer(
    new SS_LogEmailWriter('mail@mysite.com '),
    SS_Log::NOTICE
  );
}
```

Logging information

`Director::baseFolder()` fetches the file system path of your site. We use it to reference a new folder in the base directory called `logs/` where we'll put all our log files. Note that unlike most other folders, this folder isn't a module. Create the folder before referencing it and allow the web server to write to it, like `assets/`.

[You could define any folder name, but `logs` sounds like a logical choice. There is no convention behind this.]

Error levels

`SS_Log::ERR` is a constant catching all errors, `SS_Log::WARN` does the same for warnings, and you can guess what `SS_Log::NOTICE` does. But what's the difference between the different levels?

- An **error** is the most severe problem you can run into. The system can't continue with the particular operation and must abort it. An error message will be displayed to the user.
- A **warning** is when something unexpected happens. The code will try to continue its execution but a developer needs to look into this.
- A **notice** informs a developer of some minor problem which shouldn't really affect the execution of the current code. This is the least severe level of error.

When in doubt, err on the side of over-reporting problems. This makes debugging later on much easier.

SilverStripe and PHP errors

So the first three statements of our code define instructions to log all SilverStripe errors, warnings, and notices to the file `logs/ss.log`. This file will only contain your application-specific messages. This means that the PHP code itself is fine, but some problem is encountered during SilverStripe's execution. You will need to make other arrangements for logging errors within the PHP code.

The next two lines enable logging of PHP errors. These errors are logged in `logs/php.log`. Only once the PHP code is okay, can the SilverStripe framework kick in.

The layers your application uses are:

1. Network: The network on the client-side and server-side must obviously work correctly for both sides to communicate successfully. This is the most basic layer (if you ignore the client-side).
2. Operating system: The next layer is the server's operating system, which receives requests, hands them over to the webserver, and forwards its responses.
3. Webserver: The webserver (often Apache HTTP, Microsoft's IIS, or lighttpd) can respond directly to requests for static files, such as pure HTML, CSS, JavaScript, images, and so on. PHP files need to be interpreted by the web server's PHP processor module.

4. PHP: PHP is really the starting point of this book. We'll assume everything below this layer is working correctly. Using the configuration options above you'll log all errors of this layer in `logs/php.log`.
5. SilverStripe: SilverStripe itself is the top layer, requiring all the pieces below to work before being able to work its magic. As we've already said, any errors of this layer will go to `logs/php.log`.

[While we won't be able to cover all the layers, you can find detailed installation instructions in the appendix, which should help you get started.]

While we're at it, there's one more important detail: Basically there are two "classes" (not to be mixed up with PHP classes) of errors—parse and runtime:

- **Parse errors**: Every programming language has a number of rules you need to follow, called syntax. Syntax in a natural language is how sentences can be constructed. A parse or syntax error in PHP would be a missing curly bracket or semicolon for example:

```
if($var)
  echo 'This is true'
}
```

- **Runtime errors**: Once the syntax of a program is correct, there can still be semantic errors. This means that there's something wrong in the logic of your code. This happens if you try to load a nonexistent file or database, if there's a logical flaw in your code, and so on. Take a look at the following example which is all fine, unless the input is zero:

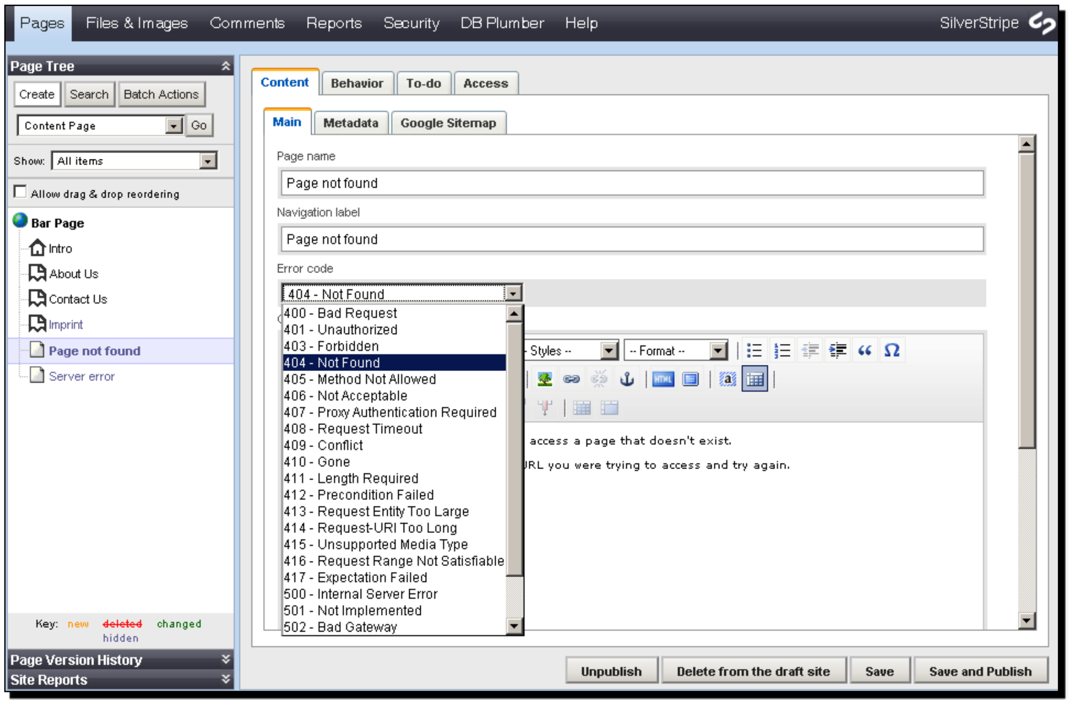
```
$numerator = 100;
$result = $numerator / $_GET['input'];
```

Website errors

On live systems you should provide error messages, which can be easily created in the CMS:

1. **Create** a new **Error Pag**e in the **Page Tree** on the left-hand side.
2. Select the **Error code** for which your custom error page should be displayed.

3. Create error pages for **404—Not Found** and **500—Internal Server Error**. The first one is used when you try to access a page that doesn't exist (probably because it never has, or you followed an incorrect or outdated link, or for other reasons), while the second one happens if the server fails to process a request (probably because of a PHP error).



Content Page

Show: All items

Page name
Page not found

Navigation label
Page not found

Error code
404 - Not Found

400 - Bad Request
401 - Unauthorized
403 - Forbidden
404 - Not Found
405 - Method Not Allowed
406 - Not Acceptable
407 - Proxy Authentication Required
408 - Request Timeout
409 - Conflict
410 - Gone
411 - Length Required
412 - Precondition Failed
413 - Request Entity Too Large
414 - Request-URI Too Long
415 - Unsupported Media Type
416 - Request Range Not Satisfiable
417 - Expectation Failed
500 - Internal Server Error
501 - Not Implemented
502 - Bad Gateway

access a page that doesn't exist.
URL you were trying to access and try again.

In `live` and `test` mode these error pages will be used, while `dev` sites will output the error message directly. This is both more friendly and informative to regular users, while malicious users won't be able to gain any information, which could be used in future attacks.

> Obviously SilverStripe can't rely on a fully functional system to generate such a message. Therefore error messages are automagically generated and saved as plain HTML in the file `assets/error-404.html` for example. So even if the database has died or you have got a serious error in your PHP code, error pages can be displayed. Note that URLs are absolute and not relative (`http://localhost/about-us` for example), so you can't copy static error pages from your development system to your server.

Notification of errors

Let's return to our example configuration from above. If your site is in live mode, all SilverStripe errors, warnings, and notices are mailed to `mail@mysite.com`. This guarantees that you'll be quickly alerted if something goes wrong on your page. Why are we only enabling this on live sites? In development mode the message is displayed to a developer anyway. This assumes that only developers are accessing development pages, so there's no point in mailing it as well. Besides, it's logged for later reference by the first three lines anyway.

Securing the logs

To wrap this part up, be sure to put a `.htaccess` file in the `logs/` folder with the following content:

```
<Files *>
  Order deny,allow
  Deny from all
  Allow from 127.0.0.1
</Files>
```

As long as you have this in a folder, no file is accessible except from `localhost`.

Go ahead, try to access `/logs/ss.log` (make sure the file already exists or create it yourself). Before creating the `.htaccess` file, the log is happily displayed. Afterwards an error page pops up, which you've already created, right?

Again, this ensures that no one else can access your log files, which may contain sensitive information or give away hints to an attacker.

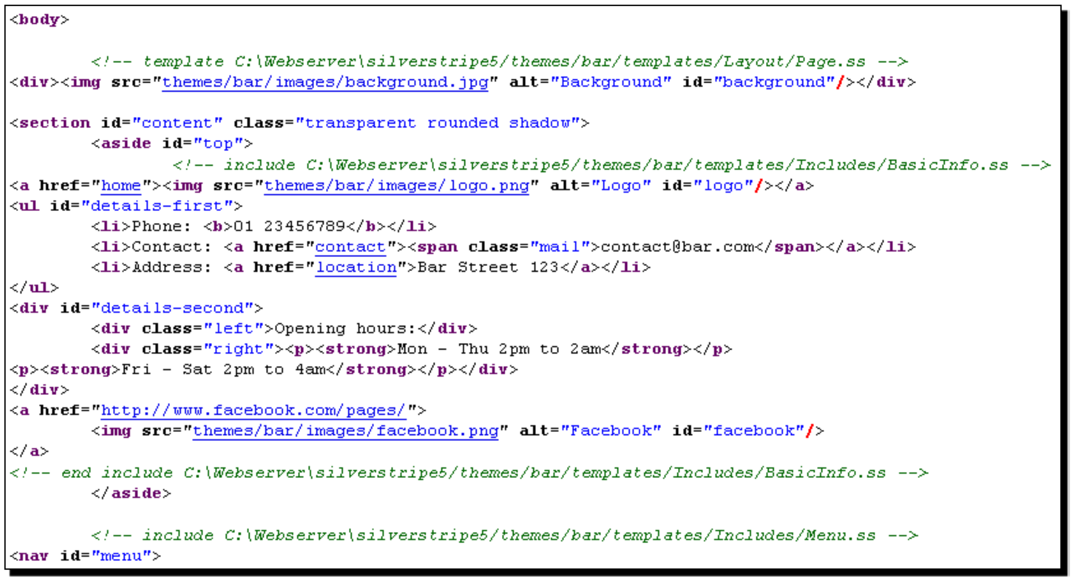
Templates

You can use the following line to remove the `m` parameter from referenced CSS and JavaScript files. If you remember what happened in the second chapter, it attached a timestamp to static CSS and JavaScript to avoid browser caching of outdated files. It's a global setting, you can't do it just for some files. Generally you should be better off without changing this option, but you might want to consider it for special cases.

```
Requirements::set_suffix_requirements(false);
```

If you're having issues with templates and includes, you can display the source files as comments in the HTML. Activate with:

```
SSViewer::set_source_file_comments(true);
```



```
<body>

        <!-- template C:\Webserver\silverstripe5/themes/bar/templates/Layout/Page.ss -->
<div><img src="themes/bar/images/background.jpg" alt="Background" id="background"/></div>

<section id="content" class="transparent rounded shadow">
        <aside id="top">
                <!-- include C:\Webserver\silverstripe5/themes/bar/templates/Includes/BasicInfo.ss -->
<a href="home"><img src="themes/bar/images/logo.png" alt="Logo" id="logo"/></a>
<ul id="details-first">
        <li>Phone: <b>01 23456789</b></li>
        <li>Contact: <a href="contact"><span class="mail">contact@bar.com</span></a></li>
        <li>Address: <a href="location">Bar Street 123</a></li>
</ul>
<div id="details-second">
        <div class="left">Opening hours:</div>
        <div class="right"><p><strong>Mon - Thu 2pm to 2am</strong></p>
<p><strong>Fri - Sat 2pm to 4am</strong></p></div>
</div>
<a href="http://www.facebook.com/pages/">
        <img src="themes/bar/images/facebook.png" alt="Facebook" id="facebook"/>
</a>
<!-- end include C:\Webserver\silverstripe5/themes/bar/templates/Includes/BasicInfo.ss -->
        </aside>

        <!-- include C:\Webserver\silverstripe5/themes/bar/templates/Includes/Menu.ss -->
<nav id="menu">
```

Most likely you'll only want to add this information during development as it's redundant information and might reveal just a little bit too much of your site's internal workings.

Hash links, also known as page anchors, (`<a href="#xxx">`)are often used in AJAX applications. By default SilverStripe will make normal links out of them (`<a href="/#xxx">`), often breaking them. To avoid this, use the following option:

```
SSViewer::setOption('rewriteHashlinks', false);
```

One more thing that could be handy is not needing to manually append the `?flush=1` if you change something in the templates during development. The following code snippet simply adds the flushing statement implicitly whenever you load a page. This adds a performance penalty as you'll never use the cache, but it can be quite convenient during development:

```
if(Director::isDev()){
  SSViewer::flush_template_cache();
}
```

Security

If you've forgotten your admin login, simply add the following statement to create a new admin user. But make sure only the right people can write to `mysite/_config.php` so this won't be used against you.

```
Security::setDefaultAdmin('admin', 'password');
```

Enforcing a password policy with SilverStripe is also pretty simple:

```
$passwordValidator = new PasswordValidator();
$passwordValidator->minLength(7);
$passwordValidator->checkHistoricalPasswords(2);
$passwordValidator->characterStrength(3, array('lowercase',
'uppercase', 'digits', 'punctuation'));
Member::set_password_validator($passwordValidator);
```

`minLength()` defines the minimum password length (in our example seven) and `checkHistoricalPasswords()` validates that the user doesn't use one of the last (two) passwords again. Finally we mandate the use of characters from at least three of the groups defined in the array. So **Test123** would be okay while **test:test** wouldn't.

You could also specify how to encrypt passwords and more; take a look at the available options in `sapphire/security/Security.php`.

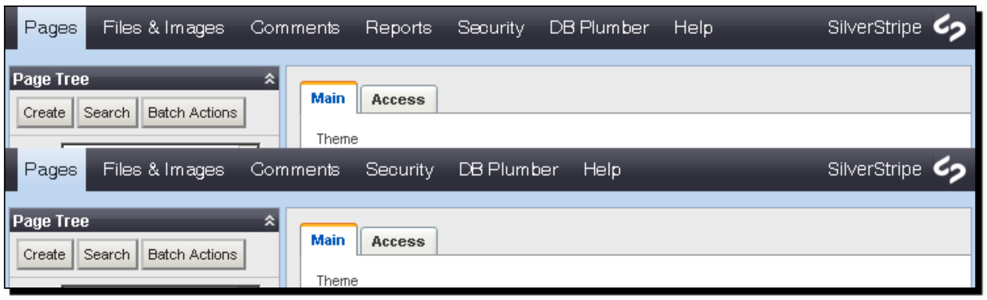
Customizing the CMS

While the CMS is already quite simple and powerful, you can further customize it to your own needs.

Removing tabs

If you don't need a tab, for example **Reports**, simply remove it:

```
CMSMenu::remove_menu_item('ReportAdmin');
```



You can find the correct string to use in the page's source code, which has HTML ID `Menu-ReportAdmin` for example. Simply remove the `Menu-` prefix and that's the string you're looking for.

Rebranding the CMS

You can also rebrand the CMS, which is allowed by the license as long as you retain the original notice in the source code.

```
LeftAndMain::setApplicationName('CMS text', 'Logo text');
```

This replaces the SilverStripe texts. Specifically the first string is used for the header title and in the footer. The second one in the top right corner next to the logo. If both should be the same, you can simply provide a single string.

```
LeftAndMain::setLogo(
  'themes/bar/images/CMSLogo.png',
  'margin: 2px;'
);
```

Use this to replace the logo in the top right corner. You can add layout information with the optional second argument.

To replace the image while the CMS is being loaded, this line comes in handy:

```
LeftAndMain::set_loading_image('themes/bar/images/CMSLoading.gif');
```

Loading add-ons

If you want to influence the layout of the CMS through CSS, or its behavior with custom JavaScript, simply add these two statements:

```
LeftAndMain::require_javascript('mysite/javascript/cms.js');
LeftAndMain::require_css('themes/bar/css/cms.css');
```

WYSIWYG

SilverStripe's WYSIWYG editor is based on the highly customizable TinyMCE. To see a demo of all options go to `http://tinymce.moxiecode.com/tryit/full.php` and check the **View Source** tab there to see how to access them.

While you can add your own CSS elements and style the content area according to your layout with `editor.css`, you can do quite a lot more in the configuration.

Switching to HTML

By default the WYSIWYG editor produces XHTML. If you use HTML4 or older (not HTML5) this may produce code that doesn't validate. To force the editor (and therefore your site) to use HTML instead, use:

```
ContentNegotiator::enable();
```

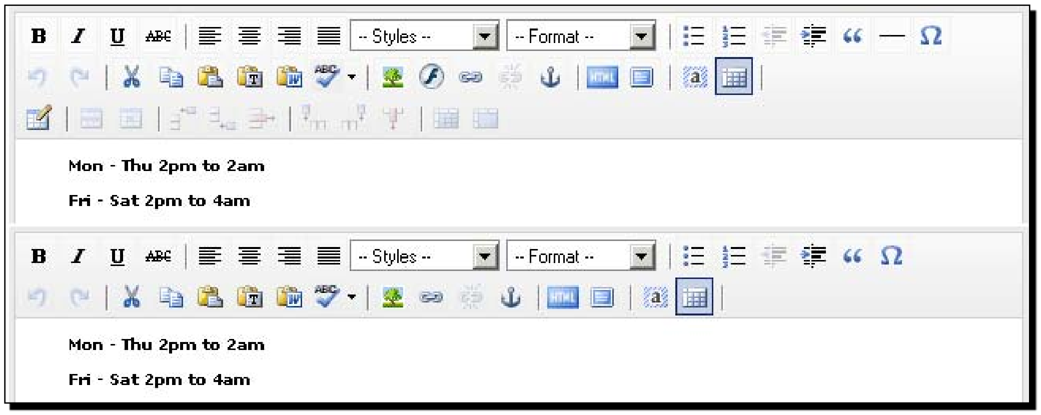
Removing buttons

If you won't need a feature you can remove it altogether. For example, if you'll never use tables in your content area, you can remove all the table buttons through `tablecontrols`.

```
HtmlEditorConfig::get('cms')->removeButtons('tablecontrols');
```

For a full list of all options, see the TinyMCE page we just referenced.

In the following screenshot, you can see the difference between the default rich text editor and the one with `tablecontrols`, horizontal rules, and Flash button removed.



Mon - Thu 2pm to 2am

Fri - Sat 2pm to 4am

Mon - Thu 2pm to 2am

Fri - Sat 2pm to 4am

Adding buttons

When adding buttons, you can decide where you want to place them. The first button is the reference to be put before or afterwards, the second is the button to be added. You can also add more than a single button in one go, by supplying more button names as further arguments.

```
HtmlEditorConfig::get('cms')->insertButtonsBefore(
  'bold', 'hr'
);
HtmlEditorConfig::get('cms')->insertButtonsAfter(
  'bold', 'sub', 'sup'
);
```

So the first line would add a button for making horizontal rules before the bold button, while the second line inserts subscript and superscript buttons after it.

You can also add to the end of the first, second or third row of buttons or replace a row altogether:

```
HtmlEditorConfig::get('cms')->setButtonsForLine(
  1, 'italic', 'bold'
);
HtmlEditorConfig::get('cms')->addButtonsToLine(
  1, 'strikethrough'
);
```

Applying the first line would remove any buttons from the first row of the editor and add `italic` and `bold`. The second one additionally adds `strikethrough`. In this case the order of the statements matters.

Configuring plugins

You can enable and disable individual plugins. For a list of bundled ones see `http://tinymce.moxiecode.com/wiki.php/Plugins/` or take a look at the folder `sapphire/thirdparty/tinymce/plugins`. Buttons are automatically added or removed when you configure them.

```
HtmlEditorConfig::get('cms')->disablePlugins('spellchecker');
HtmlEditorConfig::get('cms')->enablePlugins('searchreplace');
```

The first line removes the by default enabled spellchecker while the second one adds the search and replace functionality.

Options

Finally you can further configure TinyMCE's behavior by setting options, for more information see `http://tinymce.moxiecode.com/wiki.php/Configuration`.

If you wanted to use font tags instead of standards compliant spans you could change the default:

```
HtmlEditorConfig::get('cms')->setOption(
  'convert_fonts_to_spans', false
);
```

As you see there are hardly any limits—you can fully customize the CMS to suit your needs with a few lines of code.

Comments and spam

You can enable comments on each page in the CMS. The main problem with them is spam. SilverStripe provides some mechanisms for minimizing spam but the system is not perfect. It will just make the situation (a lot) better.

There are also additional modules, specifically `Mollom` and `Recaptcha` for increased spam protection, but we'll focus on the general configuration.

Math questions

Simply add the following line to your configuration so that you'll need to answer questions like "What is three plus two?" before being allowed to comment.

```
MathSpamProtection::setEnabled();
```

Akismet

Askismet is another popular solution. If you want to learn more about it, see `http://akismet.com`. You'll need to register on their website to use the service but it's free (at least for personal use) and pretty straight forward. You don't need to download any additional software for this to work.

After registration you'll get a `key` which you'll need to enable the service:

```
SSAkismet::setAPIKey('key');
```

Additionally you can save spam to the database so you'll be able to review it later. Append `?showspam==1` to the URL of a page with comments to review messages marked as spam after enabling this option:

```
SSAkismet::setSaveSpam(true);
```

Other measures

You can require users to have a valid user account before being able to post any messages:

```
PageCommentInterface::set_comments_require_login(true);
```

And if you want to make sure there is absolutely no spam, you can enable moderation of comments. Before a comment is displayed to any non-admin user it must be approved in the CMS in the **Comments** tab:

```
PageComment::enableModeration();
```

E-mail

It is common to build features on your website that result in e-mails being sent, such as to thank people for signing up for a newsletter. It can be complex to test whether e-mails are being sent out or have the right contents in them, but SilverStripe provides some tools to help. There's a special `Email` class which will send all e-mails for you—we'll cover that in detail later. You can simply hook into that to configure it.

You can CC or BCC any e-mail sent through the `Email` class to another address as follows:

```
Email::cc_all_emails_to('mail@mysite.com');
Email::bcc_all_emails_to('mail@mysite.com');
```

Redirecting all outgoing messages to you can be handy during testing:

```
Email::send_all_emails_to('mail@mysite.com');
```

And you can define the default sender address for all emails. We'll later cover how to overwrite this when needed:

```
Email::setAdminEmail('mail@mysite.com');
```

Miscellaneous settings

Finally, here are some additional configuration settings which can come in handy.

Accessing and advertising content

If you publish your site both under `http://www.mysite.com` and `http://mysite.com` that's not ideal. But you can easily redirect all requests to the www variant:

```
Director::forceWWW();
```

Note that your page must be in live mode for this to work. So you won't run into problems on your development system on `http://localhost`.

One feature of the `GoogleSitemap` module (included in the core download) you could enable is the notification process. Every time you publish a new page or a change, Google will be notified. This will ensure that your current content is quickly added to the search engine. There is however a big disadvantage—publishing takes much longer than before as you'll need to wait on the notification process. You'll have to decide for yourself how time-critical your changes are, or if you'd rather not enable this feature.

```
GoogleSitemap::enable_google_notification();
```

Allowing file extensions

SilverStripe's file and image manager only allows a limited set of file extensions by default. To add more use the following statement. If you want to add more than one extension simply repeat the line:

```
File::$allowed_extensions[] = 'psd';
```

Be careful which file extensions you enable, as restrictions reduce the chance of inappropriate files such as PHP scripts being uploaded by users.

Customizing Breadcrumbs

Using CSS you can easily style `$breadcrumbs`. Customizing the delimiter between pages is even easier:

```
SiteTree::$breadcrumbs_delimiter = ' >> ';
```

Image quality

SilverStripe can crop and resize images—we'll get down to the details later. By default the quality of any changed image is set to 75% of the original, which remains untouched. If that is not enough for you it can be altered, for example to 95%, but always keep in mind that it's a trade-off between file size and image quality.

```
GD::set_default_quality(95);
```

JavaScript validators

By default forms include client-side validation via JavaScript, based on Prototype–not jQuery at the moment, but this should change. If you want to write your own custom validators, you'll need to disable the default to avoid clashes:

```
Validator::set_javascript_validation_handler('none');
```

Defines

Use defines wherever possible in your code. Depending on where they belong either put them into a specific file or `mysite/_config.php` to be available in all Controllers and Models. Such "magic" values should be defined as constants and not litter the code as it makes it much harder to understand and maintain it.

By convention defines should be in UPPERCASE_WITH_UNDERSCORES.

Time for action – building our own configuration

That is quite a lot of information, but it should give you a good overview of the possibilities SilverStripe provides and can come in handy as a quick reference. Now let's put it into practice. Go back to your project and set up the following options:

1. As we're still working on our site, put it into development mode.
2. For gracefully handling our `mysite/` folder, create a constant for it. Name it `PROJECT_DIR` for keeping it consistent with the other SilverStripe defined constants.
3. Set up a database connection for the live and for the development database. Either by creating a `_ss_environment.php` file or by using `if(Director::isLive())`.
4. Set up logging for SilverStripe errors, warnings and notices as well as PHP errors.
5. If you're not in development mode, send all SilverStripe messages through e-mail to yourself. Don't use `if(Director::isDev())` as we'll put the page into test mode later.
6. Redirect all page requests to `www`.
7. Set the quality of generated images to 85%.
8. Define a constant `EMAIL` but assign different values to it in live and the other two modes.
9. Use `EMAIL` as the sender for all outgoing e-mails.
10. In development mode, send all e-mails to `EMAIL`.
11. Disable SilverStripe's JavaScript validators. We'll later write our own.
12. In development (or test) mode, flush the cache for every request.
13. If you're not in live mode, show the underlying files of templates and includes in the page's source code.

If you can't remember some of the code or how to do a task in general, just go back a few pages and take a look. Everything you'll need has been covered. Please pay attention to the mode settings or you'll have to change them again later.

Managing your code

You may be wondering if it isn't extremely tiresome to configure your application by code. Wouldn't a simple database driven graphical user interface be much quicker and simpler?

When you're doing it for the first time then yes. But if you've already set up some sites you'll greatly appreciate this approach. Simply copy the code you need, adapt it for the current project and you're good to go. If the configuration was database driven, you'd need to do the same clicks over and over again.

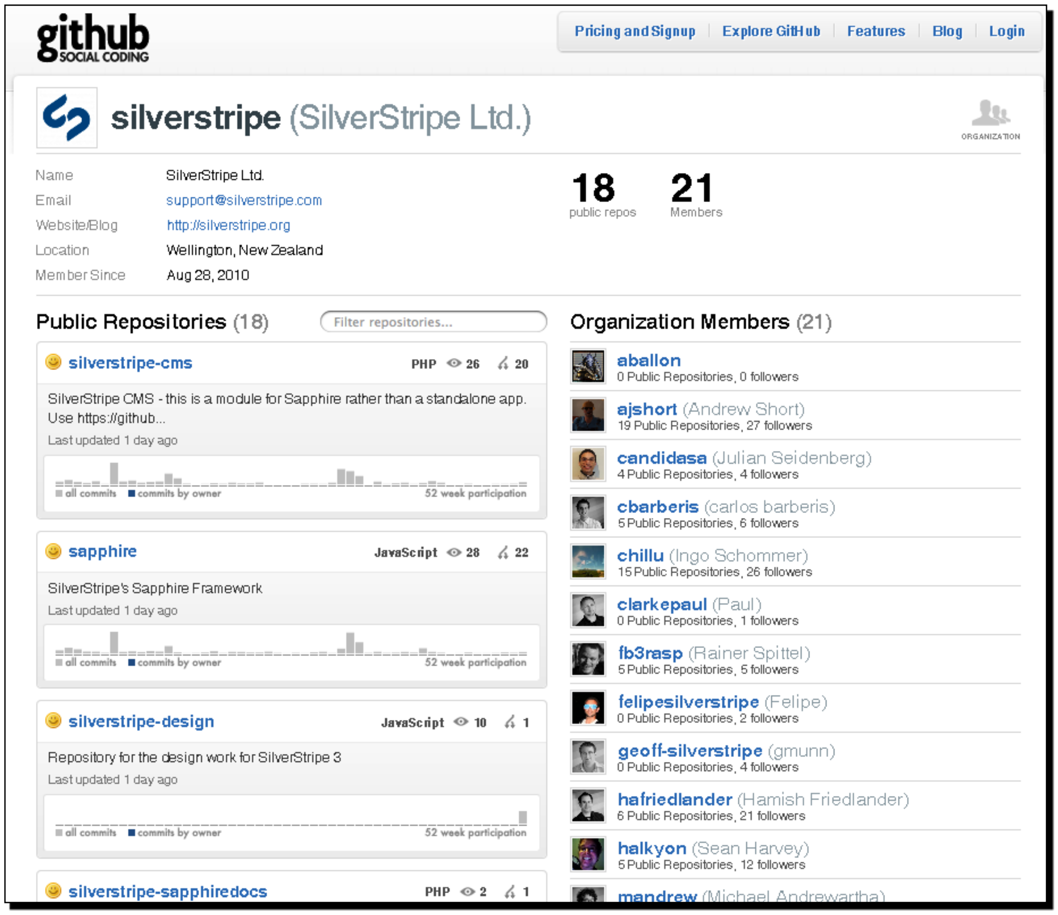
This is especially time consuming and error prone when you're extending an existing site. Assume there is a live system and you're working on your development system. After weeks of development and testing, the client approves your changes and you need to get them to the live site. How easy do you think is it to forget some configuration you did days or even weeks ago? Simply copying the database is also not an option since the live site is constantly updated. A file-based approach is much simpler in such a situation as you'd only need to copy the modified files to the live site.

Additionally by using comments you're also able to document why you chose to configure your system this way, which alternatives there might be, and what to avoid.

Version control system

Ideally you're managing your files in a **Version Control System** (**VCS**). It allows you to take snapshots of the current state, efficiently share the development of a system between multiple people, see changes and roll them back, merge conflicts, and much more. At the moment the two most widely used systems are **Subversion** (**SVN**) and **Git**. If you're new to this concept, you should definitely try to fill that gap, but we won't be able to cover it here in detail.

SilverStripe has been using Subversion for a long time, but has just recently switched to Git. Specifically the code is hosted at GitHub, which you can find at `https://github.com/silverstripe`. There you can also see the current development process and history, including every single change.



silverstripe (SilverStripe Ltd.)

Name	SilverStripe Ltd.
Email	support@silverstripe.com
Website/Blog	http://silverstripe.org
Location	Wellington, New Zealand
Member Since	Aug 28, 2010

18 public repos
21 Members

Public Repositories (18)

Filter repositories...

silverstripe-cms PHP 26 20
SilverStripe CMS - this is a module for Sapphire rather than a standalone app. Use https://github...
Last updated 1 day ago
all commits commits by owner 52 week participation

sapphire JavaScript 28 22
SilverStripe's Sapphire Framework
Last updated 1 day ago
all commits commits by owner 52 week participation

silverstripe-design JavaScript 10 1
Repository for the design work for SilverStripe 3
Last updated 1 day ago
all commits commits by owner 52 week participation

silverstripe-sapphiredocs PHP 2 1

Organization Members (21)

aballon
0 Public Repositories, 0 followers

ajshort (Andrew Short)
19 Public Repositories, 27 followers

candidasa (Julian Seidenberg)
4 Public Repositories, 4 followers

cbarberis (carlos barberis)
5 Public Repositories, 6 followers

chillu (Ingo Schommer)
15 Public Repositories, 26 followers

clarkepaul (Paul)
0 Public Repositories, 1 followers

fb3rasp (Rainer Spittel)
5 Public Repositories, 5 followers

felipesilverstripe (Felipe)
0 Public Repositories, 2 followers

geoff-silverstripe (gmunn)
0 Public Repositories, 4 followers

hafriedlander (Hamish Friedlander)
6 Public Repositories, 21 followers

halkyon (Sean Harvey)
5 Public Repositories, 12 followers

mandrew (Michael Andrewartha)

VCS basics

Let's quickly cover some of the most important terms, so that you won't get lost in the following section.

A **repository** is where current files and their historical data are being stored. In a centralized VCS (Subversion for example) there is a single, central repository. You can make a **check-out** of the repository to create a **working copy**—the place where the actual work is done. Once you're satisfied with the results of your working copy, you can **commit** your changes back into the repository. To get the latest version from a repository since your initial check-out, you can issue an **update**.

Distributed revision control systems (such as Git) have many repositories; in fact each working copy is itself a repository. Repositories can be synchronized by exchanging **patches** (change sets).

Besides this general workflow, there are often three different versions of a single repository available and SilverStripe follows this approach:

- **Tag**: Whenever a major or minor version of SilverStripe is officially released, it's also made available as a tag. This is the conservative approach and recommended for live sites.
- **Trunk** / **Master**: This is where the real development happens and all the latest commits go. You'll get the code first, but it's the most likely to break your page. This isn't recommended for live sites.
- **Branch:** A branch is the split-off from the rest of the code, allowing the development of a special version in parallel to the rest of the code. This is often used to add a new feature, which can't be developed without having an impact on the rest of the code. Once it's finished, it is merged back into the rest of the code.

Using Git in your project

As the SilverStripe core is developed with Git, it would be a good match to use it for our project as well. However it's not necessary for the following parts. We'll discuss how to get started but you can also continue without it.

[For a detailed introduction and description of Git, `http://gitref.org` is highly recommended.]

Besides the client software you'll also need the repository. You could either set up one yourself (can be a bit tricky) or use one available over the Internet. There are many free or cheap services you can use. One very popular example obviously is `https://github.com`, which is especially popular with open source projects, but there are also (paid) private plans for your own projects.

Assuming you're using a VCS and you've successfully set up your own repository, there are three approaches to working with SilverStripe:

- You download the latest stable release at `http://silverstripe.org/stable-download/` and commit it, together with your own code, into your repository. You won't need to manually copy changes between your development and live system, but you can instead do an update. However, if you want to add core updates, you'll need to manually integrate them into your repository. This method works with any VCS and not just Git or Subversion.

- If you don't want to manually integrate core updates into your repository, you can integrate the official core repositories into your own.

> The core files are split into multiple repositories for maximum flexibility. At the very least you'll need the installer (`http://github.com/silverstripe/silverstripe-installer`), Sapphire (`http://github.com/silverstripe/sapphire`), and the CMS (`http://github.com/silverstripe/silverstripe-cms`). And if you want to use SilverStripe's default theme Black Candy you'll also need `https://github.com/silverstripe-themes/silverstripe-blackcandy`.

- If you want to commit patches to SilverStripe, or want to make changes to the core files and reuse them in multiple projects, you'll need to fork these core repositories. For example if you only want to make changes to the CMS module, you'd only need to fork this one and you could use the official repositories for the others.

> A fork copies an existing repository, but makes it easy to share patches between the original and the forked project. This is necessary to get write access to the core files (in your own repository) as only SilverStripe developers have that permission on the official repository.

To make it easier to get started, SilverStripe's installer module includes a `tools/` directory that does the hard work for you. Simply create a directory for your new installation and put the installer file into it—this includes the stripped down `assets/`, `mysite/` and `themes/` folders besides the files directly in the base folder.

> Please note that the installer is very new and likely to change in the future. If you're having any issues, take a look at the official documentation, which should always be up-to-date. It's located at `http://doc.silverstripe.org/sapphire/en/installation/from-source`.

Manual upgrade of core files

For approach one, you can either download the desired version of the missing projects from the official release or you can fetch the current master through the provided "tools". To do this, open up your shell and change to the `tools/` directory of your new installation. Use the following command to download the required files:

```
new-project -m flat
```

Linux and Mac users should use `tools/new-project` while Windows users will need to work with `tools/new-project.bat`.

You can then commit all the files into your repository and start the installation of your site. As we've already said, there's no automated upgrade—you'll need to manually fetch the latest files and overwrite the existing ones.

Automatic upgrade of core files

For point two you'll need another tool called Piston, `https://github.com/francois/piston`. However, its installation goes too far for this book and has nothing to do with SilverStripe anymore. To point you in the right direction, you'll need Ruby and the Rubygems, `http://rubygems.org`, package manager to install it.

The only change to the previous section is to modify your command to:

```
new-project
```

This will fetch you a working copy instead of the latest files only, which you can then update through Piston or Git.

Contributing back

For the last approach there is no automatic solution (yet). You'll need to use plain Git commands as the tool can't guess the location of your own, forked repository. `http://gitref.org` should help you getting started.

Pop quiz – environment types

Which of the following statements are true for the three modes you can put SilverStripe into:

1. After the installation your page is automatically put into development mode, so you can easily debug problems.
2. Development sites should be in development mode, live sites should be in live mode with logging enabled.
3. You can use `Debug::show()` and `Debug::message()` in all three modes to directly output some information into your browser.
4. You need to be logged in as an administrator to call `/dev/build` and `?flush=all` on non-development sites.

Summary

In this chapter, we've taken a detailed look at SilverStripe's configuration.

Specifically, we've covered the three environments you can put your site into and what implications they have. For our example site we added the most important configuration options, in particular logging errors and mailing them to yourself (if required). Additionally we took a look at the different security options and also how to customize the CMS in general and the rich text editor specifically.

We've also discussed how SilverStripe manages its source code and how you can take advantage of this.

Now that we've learned about all these options, we're ready to make our example system more lively with widgets and short codes, which is the topic of the next chapter.

6
Adding Some Spice with Widgets and Short Codes

With all the hard work we put into the basics, it's time to do some fun stuff. After all, with SilverStripe , once you've understood the basic principles, the rest is easy.

We'll focus on four things in this chapter that allow us to build dynamic components on websites, with emphasis on the first two:

- Widgets
- Short codes
- Caching
- Text Parsers

It doesn't sound like much, but there is quite a lot you can accomplish with them—as you'll see shortly.

Why can't we simply use templates and `$Content` to accomplish the task?

- Widgets and short codes generally don't display their information directly like a placeholder does
- They can be used to fetch external information for you—we'll use Google and Facebook services in our examples
- Additionally they can aggregate internal information—for example displaying a tag cloud based on the key words you've added to pages or articles

Widget or short code?

Both can add more dynamic and/or complex content to the page than the regular fields. What's the difference?

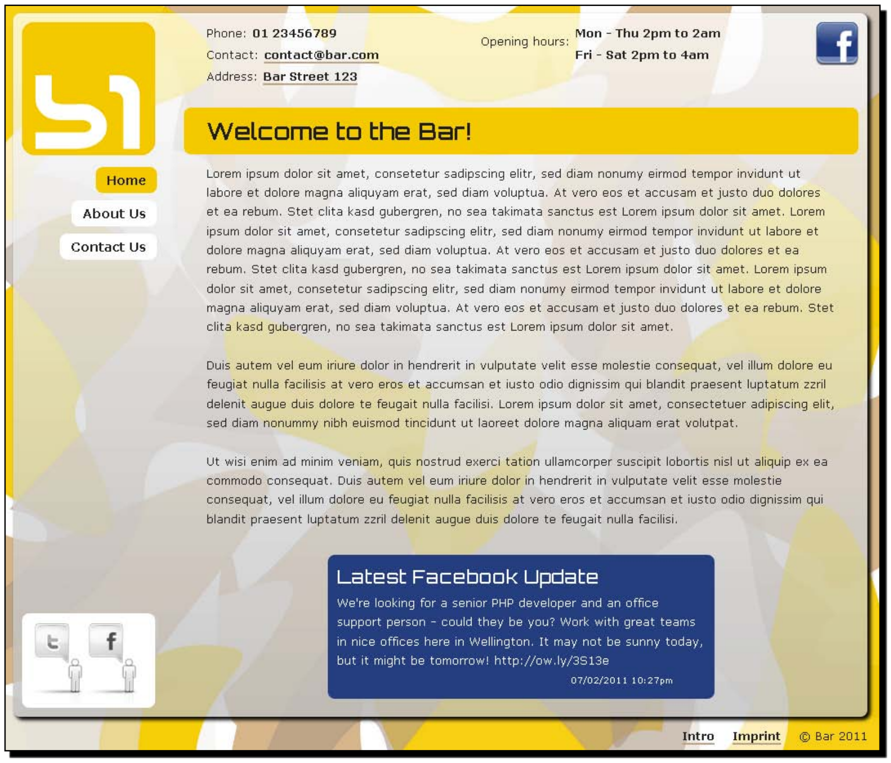
Widgets are specialized content areas that can be dynamically dragged and dropped in a predefined area on a page in the CMS. You can't insert a widget into a rich-text editor field, it needs to be inserted elsewhere to a template. Additionally widgets can be customised from within the CMS.

Short codes are self-defined tags in squared brackets that are entered anywhere in a content or rich-text area. Configuration is done through parameters, which are much like attributes in HTML.

So the main difference is where you want to use the advanced content.

Creating our own widget

Let's create our first widget to see how it works. The result of this section should look like this:



Phone: 01 23456789
Contact: contact@bar.com
Address: Bar Street 123

Opening hours: Mon - Thu 2pm to 2am
Fri - Sat 2pm to 4am

Welcome to the Bar!

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Latest Facebook Update

We're looking for a senior PHP developer and an office support person - could they be you? Work with great teams in nice offices here in Wellington. It may not be sunny today, but it might be tomorrow! http://ow.ly/3S13e

07/02/2011 10:27pm

Time for action – embracing Facebook

Facebook is probably the most important communication and publicity medium in the world at the moment. Our website is no exception and we want to publish the latest news on both our site and Facebook, but we definitely don't want to do that manually.

You can either transmit information from your website to Facebook or you can grab information off Facebook and put it into your website. We'll use the latter approach, so let's hack away:

1. In the `Page` class add a relation to the `WidgetArea` class and make it available in the CMS:

```
public static $has_one = array(
  'SideBar' => 'WidgetArea',
);

public function getCMSFields(){
  $fields = parent::getCMSFields();
  $fields->addFieldToTab(
    'Root.Content.Widgets',
    new WidgetAreaEditor('SideBar')
  );
  return $fields;
}
```

2. Add `$SideBar` to `templates/Layout/Page.ss` in the theme directory, wrapping it inside another element for styling later on (the first and third line are already in the template, they are simply there for context):

```
  $Form
  <aside id="sidebar">$SideBar</aside>
</section>
```

`<aside>` is one of the new HTML5 tags. It's intended for content that is only "tangentially" related to the page's main content. For a detailed description see the official documentation at `http://www.w3.org/TR/html-markup/aside.html`.

3. Create the widget folder in the base directory. We'll simply call it `widget_facebookfeed/`.

4. Inside that folder, create an empty `_config.php` file.

5. Additionally, create the folders `code/` and `templates/`.

6. Add the following PHP class—you'll know the filename and where to store it by now. The Controller's comments haven't been stripped this time, but are included to encourage best practice and provide a meaningful example:

```
<?php

class FacebookFeedWidget extends Widget {

  public static $db = array(
    'Identifier' => 'Varchar(64)',
    'Limit' => 'Int',
  );

  public static $defaults = array(
    'Limit' => 1,
  );

  public static $cmsTitle = 'Facebook Messages';
  public static $description = 'A list of the most recent
    Facebook messages';

  public function getCMSFields(){
    return new FieldSet(
      new TextField(
        'Identifier',
        'Identifier of the Facebook account to display'
      ),
      new NumericField(
        'Limit',
        'Maximum number of messages to display'
      )
    );
  }

  public function Feeds(){

    /**
     * URL for fetching the information,
     * convert the returned JSON into an array.
     */
    $url = 'http://graph.facebook.com/' . $this->Identifier .
        '/feed?limit=' . ($this->Limit + 5);
    $facebook = json_decode(file_get_contents($url), true);
```

```
/**
 * Make sure we received some content,
 * create a warning in case of an error.
 */
if(empty($facebook) || !isset($facebook['data'])){
  user_error(
    'Facebook message error or API changed',
    E_USER_WARNING
  );
  return;
}

/**
 * Iterate over all messages and only fetch as many as needed.
 */
$feeds = new DataObjectSet();
$count = 0;
foreach($facebook['data'] as $post){
  if($count >= $this->Limit){
    break;
  }

  /**
   * If no such messages exists, log a warning and exit.
   */
  if(!isset($post['from']['id']) || !isset($post['id'] ||
      !isset($post['message'])){
    user_error(
      'Facebook detail error or API changed',
      E_USER_WARNING
    );
    return;
  }

  /**
   * If the post is from the user itself and not someone
   * else, add the message and date to our feeds array.
   */
  if(strpos($post['id'], $post['from']['id']) === 0){
    $posted = date_parse($post['created_time']);
    $feeds->push(new ArrayData(array(
    'Message' => DBField::create(
        'HTMLText',
        nl2br($post['message'])
```

```
                ),
                'Posted' => DBField::create(
                  'SS_Datetime',
                  $posted['year'] . '-' .
                  $posted['month'] . '-' .
                  $posted['day'] . ' ' .
                  $posted['hour'] . ':' .
                  $posted['minute'] . ':' .
                  $posted['second']
                ),
              )));
              $count++;
            }
          }
          return $feeds;
        }
```

7. Define the template, use the same filename as for the previous file, but make sure that you use the correct extension. So the file `widget_facebookfeed/templates/FacebookFeedWidget.ss` should look like this:

```
<% if Limit == 0 %>
<% else %>
  <div id="facebookfeed" class="rounded">
    <h2>Latest Facebook Update<% if Limit == 1 %>
    <% else %>s<% end_if %></h2>
    <% control Feeds %>
      <p>
        $Message
        <small>$Posted.Nice</small>
      </p>
      <% if Last %><% else %><hr/><% end_if %>
    <% end_control %>
  </div>
<% end_if %>
```

8. Also create a file `widget_facebookfeed/templates/WidgetHolder.ss` with just this single line of content:

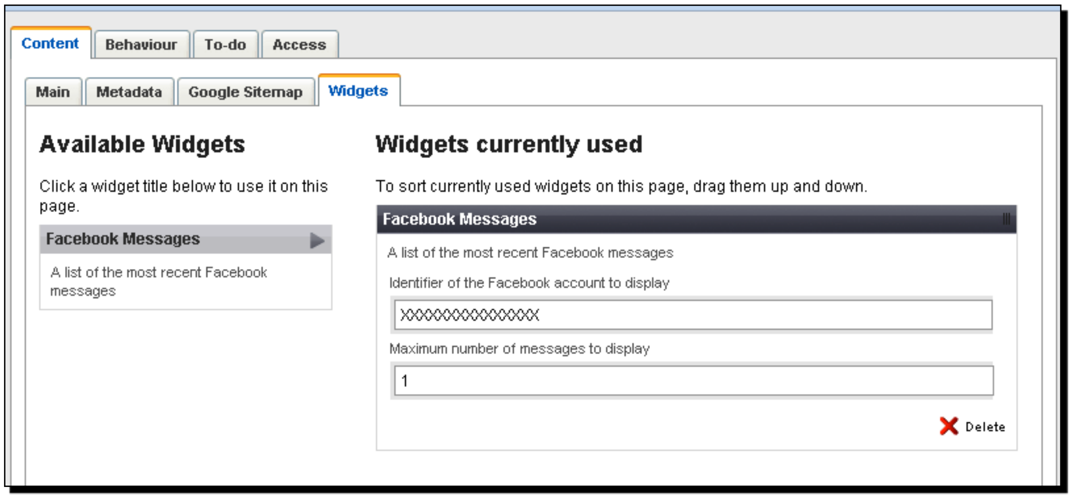
```
$Content
```

We won't cover the CSS as it's not relevant to our goal. You can either copy it from the final code provided or simply roll your own.

9. Rebuild the database with `/dev/build?flush=all`.

10. Log into `/admin`. On each page you should now have a **Widgets** tab that looks similar to the next screenshot. In this example, the widget has already been activated by clicking next to the title in the left-hand menu.

> If you have more than one widget installed, you can simply add and reorder all of them on each page by drag-and-drop. So even novice content editors can add useful and interesting features to the pages very easily.



Available Widgets

Click a widget title below to use it on this page.

Facebook Messages

A list of the most recent Facebook messages

Widgets currently used

To sort currently used widgets on this page, drag them up and down.

Facebook Messages

A list of the most recent Facebook messages

Identifier of the Facebook account to display: XXXXXXXXXXXXXXXX

Maximum number of messages to display: 1

11. Enter the Facebook ID and change the number of messages to display, if you want to.

12. **Save and Publish** the page.

13. Reload the page in the frontend and you should see something similar to the screenshot at the beginning of this section.

> `allow_url_fopen` must be enabled for this to work, otherwise you're not allowed to use remote objects such as local files. Due to security concerns it may be disabled, and you'll get error messages if there's a problem with this setting. For more details see `http://www.php.net/manual/en/filesystem.configuration.php#ini.allow-url-fopen`.

What just happened?

Quite a lot happened, so let's break it down into digestible pieces.

Widgets in general

Every widget is actually a module, although a small one, and limited in scope. The basic structure is the same: residing in the root folder, having a `_config.php` file (even if it's empty) and containing folders for code, templates, and possibly also JavaScript or images. Nevertheless, a widget is limited to the sidebar, so it's probably best described as an add-on. We'll take a good look at its bigger brother, the module, a little later.

You're not required to name the folder `widget_*`, but it's a common practice and you should have a good reason for not sticking to it.

Common use cases for widgets include tag clouds, Twitter integration, showing a countdown, and so forth. If you want to see what others have been doing with widgets or you need some of that functionality, visit `http://www.silverstripe.org/widgets/`.

Keeping widgets simple

In general widgets should work with default settings and if there are additional settings they should be both simple and few in number. While we'll be able to stick to the second part, we can't provide meaningful default settings for a Facebook account. Still, keep this idea in mind and try to adhere to it where possible.

Facebook graph API

We won't go into details of the Facebook Graph API, but it's a powerful tool—we've just scratched the surface with our example. Looking at the URL `http://graph.facebook.com/<username>/feed?limit=5` you only need to know that it fetches the last five items from the user's feed, which consists of the wall posts (both by the user himself and others). `<username>` must obviously be replaced by the unique Facebook ID—either a number or an alias name the user selected. If you go to the user's profile, you should be able to see it in the URL.

For example, SilverStripe Inc's Facebook profile is located at `https://www.facebook.com/pages/silverstripe/44641219945?ref=ts&v=wall`—so the ID is `44641219945`. That's also what we've used for the example in the previous screenshot.

For more details on the Graph API see `http://developers.facebook.com/docs/api`.

Connecting pages and widgets

First we need to connect our pages and widgets in general. You'll need to do this step whenever you want to use widgets.

You'll need to do two things to make this connection:

1. Reference the `WidgetArea` class in the base page's Model and make it available in the CMS through `getCMSFields()`. We've already covered that part in the previous chapter.
2. Secondly, we need to place the widget in our page.

$SideBar

You're not required to call the widget placeholder `$SideBar`, but it's a convention as widgets are normally displayed on a website's sidebar. If you don't have a good reason to do it otherwise, stick to it.

You're not limited to a single sidebar

As we define the widget ourselves, we can also create more than one for some or all pages. Simply add and rename the `$SideBar` in both the View and Model with something else and you're good to go. You can use multiple sidebars in the same region or totally different ones—for example creating header widgets and footer widgets. Also, take the name "sidebar" with a grain of salt, it can really have any shape you want.

What about the intro page?

Right. We've only added `$SideBar` to the standard `templates/Layout/Page.ss`. Shouldn't we proceed and put the PHP code into `ContentPage.php`? We could, but if we wanted to add the widget to another page type, which we'll create later, we'd have to copy the code. Not DRY, so let's keep it in the general `Page.php`.

The intro page is a bit confusing right now. While you can add widgets in the backend, they can't be displayed as the placeholder is missing in the template. To clean this up, let's simply remove the **Widget** tab from the intro page. It's not strictly required, but it prevents content authors from having a field in the CMS that does nothing visible on the website. To do this, simply extend the `getCMSFields()` from the previous chapter in the `IntroPage.php` file, like this:

```
function getCMSFields() {
  $fields = parent::getCMSFields();
  $fields->removeFieldFromTab('Root.Content.Main', 'Content');
  $fields->removeFieldFromTab('Root.Content', 'Widgets');
  return $fields;
}
```

Facebook feed widget

Now that we've made the general connection, it's time to focus on our widget itself. As this is quite a bit of code, let's break it down:

Facebook output

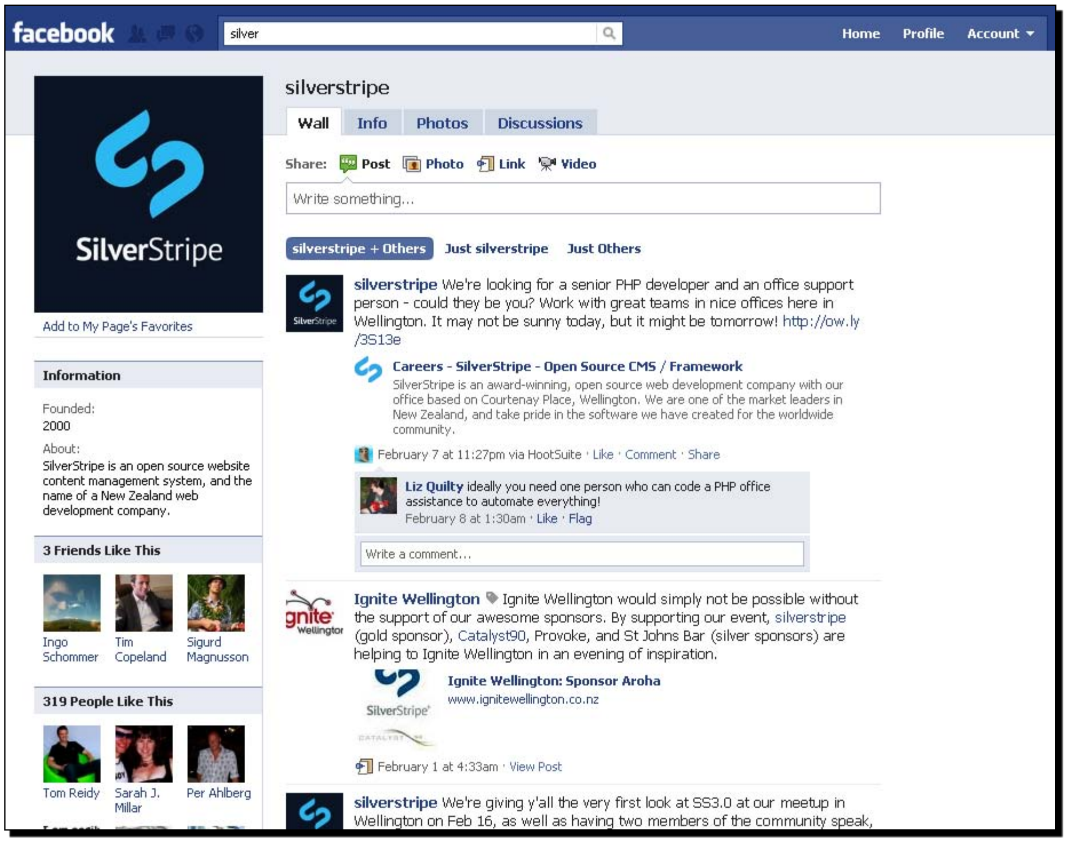
As a quick example on how the data that we receive from Facebook looks like, go to `http://graph.facebook.com/44641219945/feed?limit=2`. This fetches the last two entries of SilverStripe Inc's (public) profile.

Here's the first entry, and the start of the second one:

```
{
   "data": [
      {
         "id": "44641219945_305363039945",
         "from": {
            "name": "silverstripe",
            "category": "Product/service",
            "id": "44641219945"
         },
         "message": "We're looking for a senior PHP developer and an
office support person - could they be you? Work with great teams in
nice offices here in Wellington. It may not be sunny today, but it
might be tomorrow! http://ow.ly/3S13e",
         "picture": "http://platform.ak.fbcdn.net/www/app_full_proxy.
php?app=183319479511&v=1&size=z&cksum=8eee29030cc64ccad44483973ff24
b49&src=http\u00253A\u00252F\u00252Fsilverstripe.com\u00252Fthemes\
u00252Fsscom\u00252Fimages\u00252Flogo-small.png",
         "link": "http://silverstripe.com/about-us/careers/",
         "name": "Careers - SilverStripe - Open Source CMS /
Framework",
         "description": "SilverStripe is an award-winning, open source
web development company with our office based on Courtenay Place,
Wellington. We are one of the market leaders in New Zealand, and take
pride in the software we have created for the worldwide community.",
         "icon": "http://photos-c.ak.fbcdn.net/photos-ak-snc1/
v43/7/183319479511/app_2_183319479511_5861.gif",
         "actions": [
            {
               "name": "Share",
               "link": "http://www.facebook.com/sharer.php?u=http\
u00253A\u00252F\u00252Fsilverstripe.com\u00252Fabout-us\
u00252Fcareers\u00252F&t=Careers+-+SilverStripe+-+Open+Source+CMS+\
u00252F+Framework"
            }
         ],
         "type": "link",
         "application": {
            "name": "HootSuite",
            "id": "183319479511"
         },
         "created_time": "2011-02-07T22:27:03+0000",
         "updated_time": "2011-02-07T22:27:03+0000"
      },
      {
         "id": "44641219945_304306579945",
         "from": {
            "name": "Ignite Wellington",
            "category": "Company",
            "id": "144790912199888"
         },
```

> This is not a valid JSON document as the proper ending is missing.

The same information on the Facebook page looks like this:



silverstripe

Information

Founded:
2000

About:
SilverStripe is an open source website content management system, and the name of a New Zealand web development company.

3 Friends Like This

Ingo Schommer, Tim Copeland, Sigurd Magnusson

319 People Like This

Tom Reidy, Sarah J. Millar, Per Ahlberg

silverstripe We're looking for a senior PHP developer and an office support person - could they be you? Work with great teams in nice offices here in Wellington. It may not be sunny today, but it might be tomorrow! http://ow.ly/3S13e

Careers - SilverStripe - Open Source CMS / Framework
SilverStripe is an award-winning, open source web development company with our office based on Courtenay Place, Wellington. We are one of the market leaders in New Zealand, and take pride in the software we have created for the worldwide community.

February 7 at 11:27pm via HootSuite · Like · Comment · Share

Liz Quilty ideally you need one person who can code a PHP office assistance to automate everything!
February 8 at 1:30am · Like · Flag

Ignite Wellington Ignite Wellington would simply not be possible without the support of our awesome sponsors. By supporting our event, silverstripe (gold sponsor), Catalyst90, Provoke, and St Johns Bar (silver sponsors) are helping to Ignite Wellington in an evening of inspiration.

Ignite Wellington: Sponsor Aroha
www.ignitewellington.co.nz

February 1 at 4:33am · View Post

silverstripe We're giving y'all the very first look at SS3.0 at our meetup in Wellington on Feb 16, as well as having two members of the community speak,

The logic in general

The `Widget` class is actually a subclass of `DataObject`. This means it's automatically saved to the database in the table `Widget` and we can use the already known `$db` attribute.

First we add our two configuration options to the database—the Facebook ID and how many entries to display.

For the entries there is a sensible default value, so we use it. As we've already said, you should always provide them to make the use of your widget as easy as possible.

Then we add a title and description for our widget in the CMS so the content editors know what to do with it.

Finally we add the two configuration options to the CMS. We've done most of this before so let's jump to the new part:

```
$url = 'http://graph.facebook.com/' . $this->Identifier .
    '/feed?limit=' . ($this->Limit + 5);
$facebook = json_decode(file_get_contents($url), true);
```

- First we fetch the desired user's wall posts using `file_get_contents()`—that's why we need `allow_url_fopen`. As the wall posts include both the user's posts and posts by others on his wall, we fetch five more entries than we actually need. We'll later throw away the posts of other users, so the five additional messages are simply backup. Five may not even be enough—if you run into problems, fetch some more.
- Facebook provides the information JSON encoded. Lucky for us PHP has a function (`json_decode()`) to simply convert that into a named array.

  ```
  if(empty($facebook) || !isset($facebook['data'])){
    user_error(
      'Facebook message error or API changed',
      E_USER_WARNING
    );
    return;
  }
  ```

- To avoid any nasty errors, we check if we really got a valid response, or if there might be a problem on Facebook's side, or if the API was changed.

  Never trust another system, always check!

  In case of an error, we log the error internally (`user_error()`) and return nothing to the user. So visitors won't be able to use this specific feature, but the rest of the site will still be working as expected.

There are three kinds of errors that you can log and which are then (if configured) e-mailed to you. From the most to the least serious: `E_USER_ERROR`, `E_USER_WARNING`, `E_USER_NOTICE`. In the previous chapter, we've covered the error levels in general and also their configuration: `SS_Log::ERR` covers `E_USER_ERROR`, and so on.

```
$feeds = new DataObjectSet();
$count = 0;
foreach($facebook['data'] as $post){
  if($count >= $this->Limit){
    break;
  }

  if(!isset($post['from']['id']) || !isset($post['id']) ||
      !isset($post['message'])){
    user_error(
      'Facebook detail error or API changed',
      E_USER_WARNING
    );
    return;
  }
```

- Then we iterate over the data we got from Facebook. If we've already found enough messages, we can stop the loop. Again check for possible problems.

```
if(strpos($post['id'], $post['from']['id']) === 0){
  $posted = date_parse($post['created_time']);
  $feeds->push(new ArrayData(array(
    'Message' => DBField::create(
      'HTMLText',
      nl2br($post['message']
    ),
    'Posted' => DBField::create(
      'SS_Datetime',
      $posted['year'] . '-' .
      $posted['month'] . '-' .
      $posted['day'] . ' ' .
      $posted['hour'] . ':' .
      $posted['minute'] . ':' .
      $posted['second']
    ),
  )));
  $count++;
}
```

- Before adding a message to our output, we make sure that it was really sent by the page's owner and not someone else. We simply check that the `from.id` is equal to the start of `id` as this contains the target's ID.

  Don't rely on the `to` part of the message, there are messages which don't have it.
- If everything is okay, we add the message to the final output. The JSON doesn't include HTML entities so we need to recreate newlines through the function `nl2br()`. We also add the date when it was created—parsing it into a date and time object in the format of `YYYY-MM-DD HH:MM`. Don't worry if this isn't totally clear right now, we'll take a detailed look at `DBField::create()` in the next section.
- As the template doesn't know what to do with a raw array, it must be wrapped in a `DataObjectSet` which consists of `ArrayData` objects. These then contain arrays which are accessed in a template's control loop. After setting it up like this, it's as easy as pie to access the variables in the View.

Taking a look at the details

There are three elements in the previous code snippet that deserve special attention: Error handling, `DBField::create()`, and `DataObjectSet()`.

Error handling

There are generally two approaches to handling errors: You can either rely on the `user_error()` method:

```
user_error('Facebook detail error or API changed', E_USER_ERROR);
```

Or you can throw an exception:

```
throw new SS_HTTPResponse_Exception(
  'Facebook detail error or API changed'
);
```

Both code snippets achieve the same result, but there are some differences to consider:

- SilverStripe core uses exceptions for errors.
- Exceptions can only trigger an error. You must rely on `user_error()` to generate a warning or notice.
- If your method calls the method of another Controller, the *caller* can catch an exception of the *callee*. In our example we can't catch an exception as there's just a single method involved. If you stumble upon a block starting with `try {` and ending with `} catch(Exception $e)`, remember that this is how you catch an exception.
- If you throw an `SS_HTTPResponse_Exception()` it's automatically caught by SilverStripe and an error message is displayed.

In our example we only want a warning: Log the problem, abort the current operation, and try to load the rest of the page. So we can't use an exception.

If you feel you need a full-blown error handler and want to stop the execution of the whole page, it's a matter of choice. But as core developers are using exceptions, it makes sense to do the same.

DBField::create

Instead of passing plain strings to the View, we can use `DBField::create()` to generate database objects. This has one big advantage over strings: You can use the View's formatters (such as `$Posted.Nice` for dates) instead of manually setting it up in the Controller. This makes it way more flexible and is generally the best approach in SilverStripe.

We've already met `HTMLText` and `SS_Datetime` in the Model chapter. The first one is very simple: Simply provide a string which might contain HTML. The second one expects the format of `YYYY-MM-DD HH:MM` (following ISO 8601)—only then is the date is parsed correctly.

DataObjectSet

As the `DataObjectSet` is both often used and a bit complex, let's take a look at a minimal example:

```
public function Elements(){
  // You would need to set up the $elements array here
  $result = new DataObjectSet();
  foreach($elements as $element){
    $result->push(new ArrayData(array(
      'A' => $element['a'],
      'B' => $element['b'],
    )));
  }
  return $result;
}
```

`DataObjectSet()` is a set of DataObjects, which we've created with `DBField::create()`. Think of it as a container over which the View's `control` structure can iterate.

`ArrayData()` makes the array keys accessible as template variables. To access this example's data, your View would look like this:

```
<% control Elements %>
  $A
  $B
<% end_control %>
```

Not too hard after all, is it?

You could even do the same just using nested and associative arrays, which are automatically cast to `ArrayData()`. But we'll only use the first approach as setting the current index position is a bit cumbersome and more error prone than relying on the `push()` function.

The template

The template `FacebookFeedWidget.ss` shouldn't really use anything new, but we're using some "clever" if-else statements:

- First we're checking `<% if $Limit == 0 %>` (coming from the `$db` array in the `widget` class). Generally you'd disable the widget instead of displaying zero messages, but we're simply covering that situation in our template. As there isn't a negation we're providing an empty output for zero messages, otherwise the information box is added to the page.
- If there is more than one message, we want to display **Latest Facebook Updates** instead of **Latest Facebook Update**. Without negation, bigger and smaller comparators, we're falling back to: `<% if Limit == 1 %><% else %>s<% end_if %>`. This might be a little cumbersome, but you're getting used to it—that's the price of a very simple template language.
- The `<% if Last %>` ensures that a horizontal rule is added between messages, making sure it's added only in between messages and not after the last entry. But why are we making it so complicated—what about `<% if Middle %>`? Good idea, but that wouldn't put a line after the first element. And `<% if First || Middle %>` wouldn't work as well as this would always add a horizontal rule after the first element—even if there's just a single one.

Using the `control` structure we iterate over all messages, accessing `$Message` and `$Posted`. We've already taken a look at this in the previous paragraph. In the default case it's only a single iteration, but we're also prepared for more messages.

The other template

What is the file `WidgetHolder.ss` doing? Actually it's not really necessary for the widget to work. It only overwrites SilverStripe's default widget holder (located in `sapphire/templates/WidgetHolder.ss`) that wraps each of our widgets in some additional markup:

```
<div class="WidgetHolder $ClassName
    <% if FirstLast %> $FirstLast<% end_if %>">
  <% if Title %><h3>$Title</h3><% end_if %>
  $Content
</div>
```

And if you don't need it, we've just discussed how to replace it. We could edit this file directly, but that would be a really bad idea. We'd have to redo the change every time we update the core files. Thanks to inheritance, this is not necessary.

All we need to do is add a file with the same name to our widget's template directory and it's used instead. The `$Content` placeholder only includes our own `FacebookFeedWidget.ss`—if you leave the widget holder file empty, our custom template is ignored.

> Why is this file even there? We didn't need it and it only makes our life more complicated... Well, assume you want to wrap all your widgets in the same markup. Copying that code to every widget's template is pretty cumbersome, so it might be handy to have this container in place. And if you don't need it, we've just discussed how to avoid it.

More widgets

In our example we used Facebook as our external data source. In most cases you'll use information from inside the system using DataObjects. For example, for displaying the latest content, the newest member, and so on. This is just a hint to guide you in the right direction—you can, but you are not required to use the database when using widgets.

Additionally, widgets are very portable. We could have hard-coded all of this functionality right into our custom template, but only widgets can be easily reused on other sites. Simply copy the widget folder over, rebuild the database, and you're finished!

Have a go hero – Twitter

There are already Twitter widgets available for download. But if you want to give it a try yourself or want to include Twitter into our Facebook widget, don't be shy.

Twitter provides a pretty similar API for retrieving tweets, `<username>` must again be replaced: `http://twitter.com/statuses/user_timeline/<username>.json?count=2`

> Note that the response JSON is different. Besides changing the URL, you'll also need to slightly adapt the PHP code processing the underlying data.

If you want more details, take a look at the official documentation at `http://dev.twitter.com/doc`.

Text parser

Is the `nl2br()` in the widget's logic bothering you as well? Adding markup anywhere else than in the template is definitely not nice. Luckily for us, SilverStripe provides a handy tool for this problem.

Time for action – doing it "right"

1. Remove the `nl2br()` in the file `FacebookFeedWidget.php`, so the code looks like this:

   ```
   'Message' => DBField::create('Text', $post['message']),
   ```

2. Create a new file `mysite/code/ Nl2BrParser.php` with the following code:

   ```
   class Nl2BrParser extends TextParser {
     public function parse(){
       return nl2br(htmlentities(
         $this->content,
         ENT_NOQUOTES,
         "UTF-8"
       ));
     }
   }
   ```

3. In the template `FacebookFeedWidget.ss` replace `$Message` with `$Message.Parse(Nl2BrParser)`.

4. Flush your page for the template engine to pick up the parser—rebuilding the database is not required.

What just happened?

Obviously we've created a text parser. That's pretty straightforward: Create a new class that extends `TextParser` and implement a function `parse()`; in the template, attach `.Parse()` to the placeholder and add the parser's class name inside the brackets. The placeholder that you want to change can be accessed through `$this->content` in the parser.

There are two things you should be aware of:

- `TextParser`, as the name implies, can only work with the `Text` data type or subtypes of it, for example `HTMLText`; not on `Varchar` or a plain old string. Therefore in our example we need to make sure it has the right data type: `DBField::create('Text', $post['message'])`.

- When using your own text parser, you potentially circumvent SilverStripe's security mechanisms. Text only strings won't escape HTML automatically. This is only done by the default DBField, which we've overwritten. So we need to take care of that ourselves, for example with `htmlentities()`—stripping any markup from the text. We've done just that in our example.

Where to store the parser

Whether you want to put the parser in `mysite/code/` or into the widget's `code/` directory depends on where you want to use it. We assumed that it may come in handy for other elements besides widgets, so we used `mysite/`. If you're sure the parser will only be used in a single widget, you should store it there directly.

Other examples where text parsers can come in handy include displaying only the first X characters or words of a string, syntax highlighting, and so on.

Caching

It's great that we've added the Facebook integration, it makes our pages much more lively without adding any work for the content editors. But it's not efficient—every single time a visitor loads the page, we need to fetch the latest information from the remote server, parse it, and output it. While we don't care too much for extra load on Facebook (they can definitely handle it as we're hardly making a big difference), it's bad for our performance:

1. The widget class needs to fetch the data and wait for the response. The visitor must wait for that to finish.
2. We need to parse the response and output it, so there's more work for our server to do. And in most cases it's the same work over and over again.

Can that work be done once and the result stored? From the second visitor onwards only the pre-computed output would need to be fetched. This is a very common concept called **caching**. SilverStripe provides multiple methods for making use of it.

Partial caching

This allows the caching of parts of a page—hence partial. You can mix cached and uncached parts easily. However this also provides the least performance gain of all methods as the Controller is always needed for generating the final page. That said, in our example it removes the network latency and web service call to Facebook, so it provides a major speed increase.

We'll take a better look at it shortly.

Static publisher

Partial caching is not enough once you get lots of hits, at which point you should try out the static publisher. It generates the raw HTML output and serves it without ever touching PHP or the database. You do need to regenerate it every time something changes, but it allows you to have hundreds of page views per second.

The downside of this is that it is rather complex to set up, as every page needs to know about all potential changes. For example if you add a page to the menu, all pages displaying the menu will need to be regenerated. We won't cover the static publisher in detail. Just remember that there is more than partial caching, should you ever need it.

Static exporter

This method lets you export a copy of your website as an archive. It's the same principle as the static publisher, only that you can't select specific pages (everything is exported) and that you need to manually upload the files to your server from the archive.

We won't cover this functionality as only few people should need it, but we've mentioned it for the sake of completeness.

Restful service

`RestfulService` enables you to easily connect to web services—just what we've done with Facebook. However its main power is that it can easily parse and search XML responses. It doesn't provide too much added value for our JSON documents and that's the reason why we haven't used it.

RESTful

Representational State Transfer (**REST**) is an architectural style, and a web service is RESTful if it adheres to the REST principles. Without going into the finer details, the three aspects for REST are that a service has a URL, it communicates through HTTP, and the data is exchanged in valid Internet media types (commonly XML or JSON). Our Facebook example is a RESTful web service, we've just saved the proper definition for now.

This service doesn't only fetch a response, but can also cache it. So you don't need to query Facebook again and again, but can instead rely on your server's cached version.

Partial caching

The basic principle of partial caching is to put control structures into the template files, telling the template engine what to cache. As the Controller is always used before getting to the template, you won't gain as much as with static publishing or exporting.

Example

A basic example would look like this:

```
<% cached %>
  The cached content...
<% end_cached %>
```

Time for action – caching

Let's try that out in our code.

1. Add the following code to your `mysite/_config.php`:

   ```
   if(Director::isDev()){
     SS_Cache::set_cache_lifetime('any', -1, 100);
   } else {
     SS_Cache::set_cache_lifetime('any', 3600, 100);
   }
   ```

2. In `mysite/code/Page.php` add a new method in the Controller:

   ```
   protected function CacheSegment(){
     return $_SERVER["REQUEST_URI"];
   }
   ```

3. In `themes/bar/templates/Page.ss` wrap everything between the following two lines:

   ```
   <% cached 'page', LastEdited, CacheSegment %>
     <%-- The content of your page --%>
   <% end_cached %>
   ```

4. Switch your page into test mode and reload a page.

We'll take a look at how to check if caching is actually working after going through the code.

What just happened?

Let's break it down into the following components:

Configuration

In the configuration file we defined that caching should only be enabled in test and live mode, but not in development. So be sure to put the page into the right mode, before working on this.

`SS_Cache::set_cache_lifetime('any', 36000, 100)` sets the policy for `any` file, for a duration of one hour (time in seconds, `60 x 60 = 3600` s) with a priority of `100`. You could set different strategies with various priorities, but we won't need that. We simply don't cache at all in development mode (`-1` seconds) and cache everything for one hour in the other two modes.

We could of course make the duration much longer, but in our example this is not advisable. As we're caching the Facebook message on our page (our original goal), an update will appear after 60 minutes at most and 30 minutes on average. That sounds like a reasonable amount of time, so we'll go with that.

Controller

In the Controller we created a helper function, returning the current URL. Why didn't we use `$URLSegment`? Because it only returns the page's given URL, no actions, IDs or parameters. For finer control we'll need those later, so we create this simple helper.

Assume you have the URL `/page/action/id/?param1=a&param2=b`: `$URLSegment` returns `page`; our helper function returns `/page/action/id/?param1=a&param2=b`.

Template

In the template we define when a cached page should be used and when not, so let's break down `<% cached 'page', LastEdited, CacheSegment %>`:

- The part after `cached` is called the cache key. It consists of an unlimited number of comma separated elements. Having multiple elements allows you to have a fine-grained control over your cache strategy.
- Cache keys can either be (quoted) strings or regular placeholders as well as Controller functions.
- The values of the cache key are hashed. The system then tries to find a file-name of the hash value. If such a file is found, it is used, otherwise a new one is generated. So every time one of the elements changes, a new cache set must be created.
- Old cache files are automatically subject to garbage collection.

- First we define a name for our cache file (`page`) so that there are no collisions with other cache blocks.
- Then we require the cache to be regenerated after the current page is saved—the `$LastEdited` placeholder or database field comes in handy here.
- Finally we want one cache file per URL, using the Controller function `CacheSegment` that we created a little earlier. This makes sure that the `/home` and `/start` page don't rely on the same cache file. They display totally different content, so a common cache wouldn't work out.

General considerations

Should we use one big cache file, or many small ones instead?

It really depends, as there is always some trade-off. In general if your cache blocks are very small, you'll need fewer than with bigger blocks. Having small cache files, you can aggregate them in various combinations, whereas with bigger ones you'll need to create many, often containing the same content with minor variations.

So we should rather make the blocks as small as possible? Not really—fetching a cached file is expensive in terms of performance. You need to calculate the hash, check if the cache already exists, and possibly generate a new one. So bigger blocks with fewer checks for cache files will generally lead to better performance, at the cost of the number and size of cache files. Let's illustrate this with two simple examples. For clarity, the actual content of the different elements is simply represented by a SilverStripe comment:

One big cache block where `/home/something` results in a different "URL variation" block than `/home/else`.

```
<% cached 'page', LastEdited, CacheSegment %>
  <%-- Header --%>
  <%-- Menu --%>
  <%-- Content --%>
  <%-- URL variation --%>
  <%-- Footer --%>
<% end_cached %>
```

Many small cache blocks where header, menu, and footer sections are relatively static and identical on all pages. `/home/something` results in a different "URL variation" block than `/home/else`, but the block is identical on `/into/something`. There is a Controller method `UrlVariation()` to check for this (referenced in the cached statement).

```
<% cached 'header' %>
  <%-- Header --%>
<% end_cached %>
<% cached 'menu' %>
```

```
    <%-- Menu --%>
  <% end_cached %>
  <% cached 'content', LastEdited %>
    <%-- Content --%>
  <% end_cached %>
  <% cached UrlVariation %>
    <%-- URL variation --%>
  <% end_cached %>
  <% cached 'footer' %>
    <%-- Footer --%>
  <% end_cached %>
```

Assume we have a lot of pages such as `/home`, `/home/a`, `/home/b`, `/home/c`, `/intro`, `/intro/a`, `/intro/b`, `/intro/c`, `/about`, `/about/a`, `/about/b`, `/about/c`, and so on. The difference between the two approaches are as follows:

- The first code example runs faster than the second one. The system only needs to check one cache key and fetch a single file, whereas the second one requires five such actions.
- The first example requires more disk space as a lot of information will be duplicated (header, menu, and footer) in each cached file.
- The second approach requires fewer files: One global header, menu, and footer, one block for each first URL part (`/home`, `/intro`, `/about`) and one for each second URL part (nothing, `/a`, `/b`, `/c`), resulting in ten small files.
- The first approach requires more and bigger files: One for each unique URL. Twelve altogether for the URLs listed above, but the more pages you have the more this number will grow.

You'll need to find the best solution for yourself. As our pages are pretty simple, we'll go with big cache blocks—as you've already seen.

Finally, at the risk of stating the obvious: Caching static parts in the template (pure HTML or CSS) won't provide much benefit, there are no database queries or PHP logic to avoid. Focus on the right parts when caching!

Flushing the page does not delete the cache! As you're adding a URL parameter, a new cache file is created in our example, but the general cache file is left untouched, waiting to time out. If you really need to get rid of the cache, remove the files from the caching directory (which we'll cover in a minute).

More on partial caching

Cached blocks can be nested, as they are handled independently. This can be useful for mixing general and specific blocks.

Cached blocks can't be contained in `if` or `control` statements. If you do, you'll get an error message from the template engine. In most cases this should be avoidable as there are some advanced caching statements you can use:

- To cache a block for each user individually, use `CurrentMember.ID`. For example if you want to output the current user's first name you could cache it like this:

  ```
  <% cached 'user', CurrentMember.ID %>
    Hello $CurrentMember.FirstName!
  <% end_cached %>
  ```

- To make it dependant on the user's rights, use `CurrentMember.isAdmin`
- To require a new cache block whenever any page in the system changes, use `Aggregate(Page).Max(LastEdited)`. This takes all pages (`Aggregate(Page)`) and by taking the highest (`Max(LastEdited)`), gets the last time, any page was edited.

If you want to exclude something within an otherwise cached block, wrap it between `<% uncached %>` and `<% end_uncached %>`. You can also provide a cache key, which is of course ignored, but it is very handy to (temporarily) disabling some caching.

Taking it even further

If you need more performance, be sure to check out the API documentation. You can for example use Memcached (`http://memcached.org`) for keeping cache entries in memory. While you'll need more (expensive) memory in your server, it's much faster than reading cached files from disk.

Carefully test your cache settings

At first it sounds trivial—what could possibly go wrong?! But it can be tricky. Let's think about our (very simple) menu:

- The naive approach would possibly be to simply cache the menu for all pages, as it's identical for all pages.
- Not quite! There's a tiny bit of markup added to distinguish the currently active page: Instead of `<li class="link">` it's `<li class="current">`. So in order to mark the current page through CSS, we need to store the menu for each page, as it's always different. Is that it?

- Not quite so fast. If we change, delete, or add a page, we'll want all pages to reflect such a change immediately. So instead of only caching the menu per page, we'll also need to expire all of these cached files as soon as a single menu entry is changed.

So for our menu we could either add a nested caching statement in the menu file `themes/bar/templates/Includes/Menu.ss` or change the page-wide setting. In light of using bigger caching files, we're going with the second option, so change the original statement (which we've just described) to:

```
<% cached 'page', Aggregate(Page).Max(LastEdited), CacheSegment %>
```

Alternatively you could change your menu to:

```
<nav id="menu">
  <ul>
    <% cached 'menu', Aggregate(Page).Max(LastEdited), URLSegment %>
      <% control Menu(1) %>
        <li class="$Linkingmode">
          <a href="$Link">$MenuTitle</a>
        </li>
      <% end_control %>
    <% end_cached %>
  </ul>
</nav>
```

So we have our `menu` block. Whenever any page in the system is changed, a new cache block is required.

Very easy so far, but why do we use the old `$URLSegment` instead of our `CacheSegment()` function? In our specific case we don't need it. The menu is so simple that it only contains the main page—if there are any subpages or further arguments, the menu should be the same. So there is no point in generating multiple identical cache files.

Yes, it's a trade-off. As we've already said, consider for yourself which is the best option in your specific project.

While it's not really hard, some care is needed to find the right strategy. You'll want to remove outdated information quickly while reusing cached parts as often as possible. Even for such simple elements as our menu, it can be a little tricky.

Cache directory

Now that we know how caching works, it would be nice to know where the cached files are stored. The easy answer is: in SilverStripe's cache directory. But that can be one of three possible places:

- If you define the constant `TEMP_FOLDER`, its value is used as the cache directory. However note that you can't set this up in `mysite/_config.php`, because the constant must be set up earlier. If you want to use this approach, you need to add the following line of code to `_ss_environment.php`, which we've already covered in the configuration chapter:

  ```
  define('TEMP_FOLDER', '/full/path/to/cache/directory');
  ```

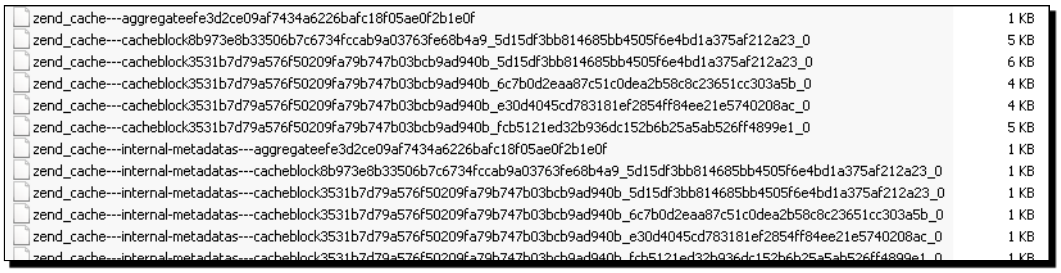
- If you didn't define `TEMP_FOLDER`, the framework checks if you have a folder `silverstripe-cache/` in your installation's base directory (just beside `sapphire/`, `cms/`, `mysite/`, and so on). If such a folder is found, it's used as the cache directory.
- If you didn't set the constant or create the folder, SilverStripe defaults to the value from the system's temporary directory and appends `silverstripe-cache` plus your installation's base directory (slashes replaced by hyphens). Assume you installation is located in `Webserver/silverstripe6/` and PHP has not been configured to specifically use a different temporary directory:
  - On Linux or Mac, the first part defaults to the `/tmp/` directory. So the cache directory would be: `/tmp/silverstripe-cache-Webserver-silverstripe6`.
  - On Windows, it's a user-specific folder, for example `C:/DOCUME~1/theuser/LOCALS~1/Temp/`—assuming the username is `theuser` and you're using an English version of Windows. So the cache directory would then be `C:/DOCUME~1/theuser/LOCALS~1/Temp/silverstripe-cacheC--Webserver-silverstripe6`—if your installation is on the `C:\` drive.

Unless you're having issues with the cache directory, there is no point in changing the default.

Setting the correct permissions

For the caching to work, it's vital that the webserver is allowed to write and read the cache directory. This is a common pitfall. Double check that this is configured correctly.

Now let's get back to our original issue: Within your cache directory, there should be a subfolder `cache/`. It should include some `zend_cache---cacheblock*` files, that have been generated by your cache control structures. Take a look at them. You'll see plain HTML and possibly some CSS or JavaScript. It's all client-side code ready to be sent to visitors without further server-side processing.



```
zend_cache---aggregateefe3d2ce09af7434a6226bafc18f05ae0f2b1e0f                                                          1 KB
zend_cache---cacheblock8b973e8b33506b7c6734fccab9a03763fe68b4a9_5d15df3bb814685bb4505f6e4bd1a375af212a23_0              5 KB
zend_cache---cacheblock3531b7d79a576f50209fa79b747b03bcb9ad940b_5d15df3bb814685bb4505f6e4bd1a375af212a23_0              6 KB
zend_cache---cacheblock3531b7d79a576f50209fa79b747b03bcb9ad940b_6c7b0d2eaa87c51c0dea2b58c8c23651cc303a5b_0              4 KB
zend_cache---cacheblock3531b7d79a576f50209fa79b747b03bcb9ad940b_e30d4045cd783181ef2854ff84ee21e5740208ac_0              4 KB
zend_cache---cacheblock3531b7d79a576f50209fa79b747b03bcb9ad940b_fcb5121ed32b936dc152b6b25a5ab526ff4899e1_0              5 KB
zend_cache---internal-metadatas---aggregateefe3d2ce09af7434a6226bafc18f05ae0f2b1e0f                                     1 KB
zend_cache---internal-metadatas---cacheblock8b973e8b33506b7c6734fccab9a03763fe68b4a9_5d15df3bb814685bb4505f6e4bd1a375af212a23_0   1 KB
zend_cache---internal-metadatas---cacheblock3531b7d79a576f50209fa79b747b03bcb9ad940b_5d15df3bb814685bb4505f6e4bd1a375af212a23_0   1 KB
zend_cache---internal-metadatas---cacheblock3531b7d79a576f50209fa79b747b03bcb9ad940b_6c7b0d2eaa87c51c0dea2b58c8c23651cc303a5b_0   1 KB
zend_cache---internal-metadatas---cacheblock3531b7d79a576f50209fa79b747b03bcb9ad940b_e30d4045cd783181ef2854ff84ee21e5740208ac_0   1 KB
zend_cache---internal-metadatas---cacheblock3531b7d79a576f50209fa79b747b03bcb9ad940b_fcb5121ed32b936dc152b6b25a5ab526ff4899e1_0   1 KB
```

Once you've found these files, you can be sure that caching is working as it should.

> In the cache directory itself (not the `cache/` subfolder) SilverStripe automatically caches parsed templates and database-related files, speeding up the whole site's speed considerably. If you delete these files, they'll be automatically regenerated, but it may take a little while.

Performance gains

The final question before finishing this section is: How big is the performance benefit when using partial caching? Let's take a look at some simple benchmarks.

> The "server" in this case was a netbook. However as many hosting providers won't provide much more power than that on their low to medium sized packages, this should be quite informative. Just don't think that SilverStripe can't go faster on more powerful machines.

There are generally two sides when doing benchmarks: The server-side (how many requests can the server handle) and the client-side (how fast is my single request). We'll take a look at both.

Server-side

For our simple benchmark, we're using **ApacheBench** (**ab**) which is part of the Apache HTTP server. It's a shell command line tool. The following example tries to fetch the given URL 50 times with five concurrent connections as fast as possible:

```
ab -n 50 -c 5 http://localhost/
```

The results of this test, done on our page including the latest Facebook messages, are (with minor variations):

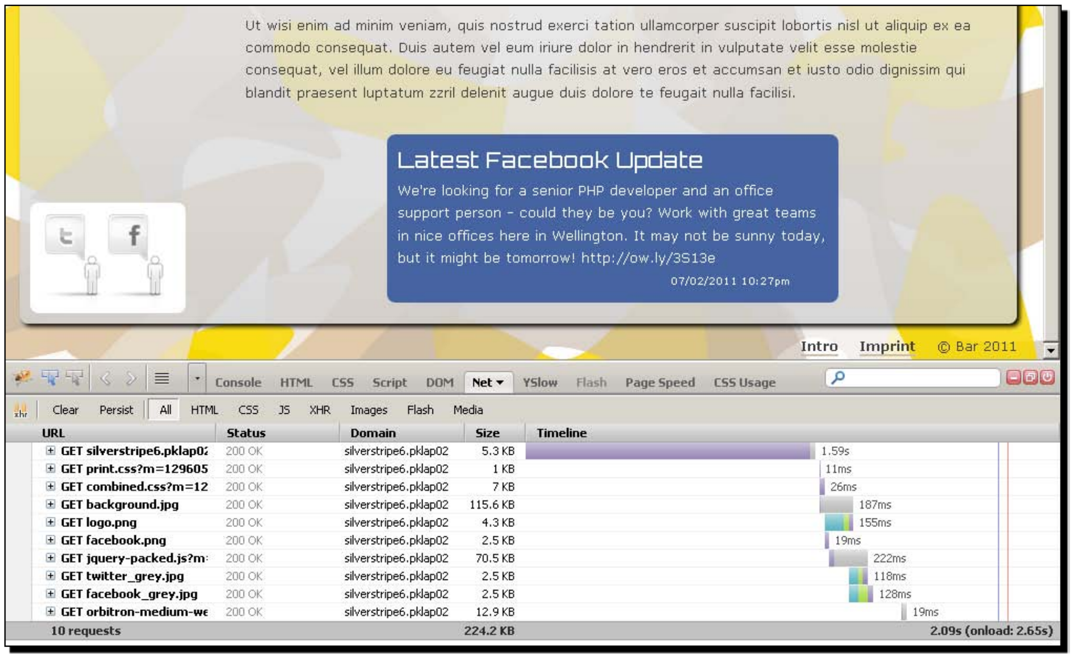
- With partial caching enabled, it took approximately 22.85 seconds, resulting in nearly 2.2 requests per second
- With partial caching disabled it took approximately 32.38 seconds, resulting in a good 1.5 requests per second

This means we could fulfill nearly 50% more requests with partial caching than without!

Client-side

To measure the time it takes to fulfill your request, you can (for example) use Firefox's excellent FireBug plugin, `http://getfirebug.com`.

While there's less difference when processing a single request than when processing many concurrent ones, the overall trend is the same. Take a look at the following screenshots—the first one was taken with partial caching enabled and the second one without it.



Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

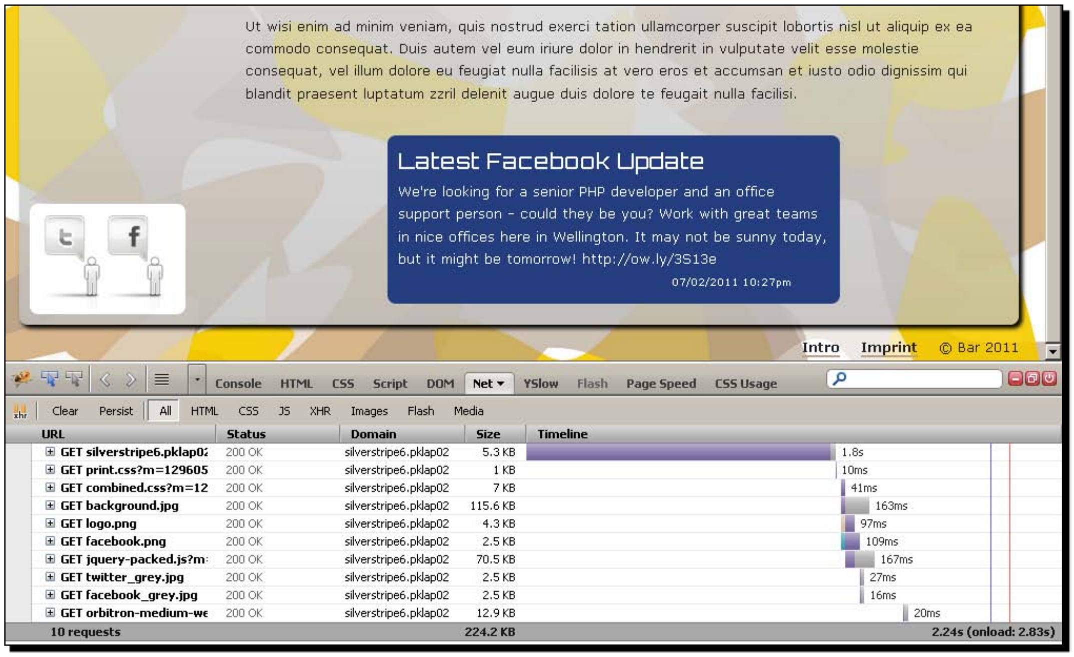
Latest Facebook Update

We're looking for a senior PHP developer and an office support person - could they be you? Work with great teams in nice offices here in Wellington. It may not be sunny today, but it might be tomorrow! http://ow.ly/3S13e

07/02/2011 10:27pm

Intro Imprint © Bar 2011

URL	Status	Domain	Size	Timeline
GET silverstripe6.pklap02	200 OK	silverstripe6.pklap02	5.3 KB	1.59s
GET print.css?m=129605	200 OK	silverstripe6.pklap02	1 KB	11ms
GET combined.css?m=12	200 OK	silverstripe6.pklap02	7 KB	26ms
GET background.jpg	200 OK	silverstripe6.pklap02	115.6 KB	187ms
GET logo.png	200 OK	silverstripe6.pklap02	4.3 KB	155ms
GET facebook.png	200 OK	silverstripe6.pklap02	2.5 KB	19ms
GET jquery-packed.js?m	200 OK	silverstripe6.pklap02	70.5 KB	222ms
GET twitter_grey.jpg	200 OK	silverstripe6.pklap02	2.5 KB	118ms
GET facebook_grey.jpg	200 OK	silverstripe6.pklap02	2.5 KB	128ms
GET orbitron-medium-we	200 OK	silverstripe6.pklap02	12.9 KB	19ms
10 requests			224.2 KB	2.09s (onload: 2.65s)



Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Latest Facebook Update

We're looking for a senior PHP developer and an office support person - could they be you? Work with great teams in nice offices here in Wellington. It may not be sunny today, but it might be tomorrow! http://ow.ly/3S13e

07/02/2011 10:27pm

URL	Status	Domain	Size	Timeline
GET silverstripe6.pklap02	200 OK	silverstripe6.pklap02	5.3 KB	1.8s
GET print.css?m=129605	200 OK	silverstripe6.pklap02	1 KB	10ms
GET combined.css?m=12	200 OK	silverstripe6.pklap02	7 KB	41ms
GET background.jpg	200 OK	silverstripe6.pklap02	115.6 KB	163ms
GET logo.png	200 OK	silverstripe6.pklap02	4.3 KB	97ms
GET facebook.png	200 OK	silverstripe6.pklap02	2.5 KB	109ms
GET jquery-packed.js?m:	200 OK	silverstripe6.pklap02	70.5 KB	167ms
GET twitter_grey.jpg	200 OK	silverstripe6.pklap02	2.5 KB	27ms
GET facebook_grey.jpg	200 OK	silverstripe6.pklap02	2.5 KB	16ms
GET orbitron-medium-we	200 OK	silverstripe6.pklap02	12.9 KB	20ms
10 requests			224.2 KB	2.24s (onload: 2.83s)

While not SilverStripe-specific, PHP accelerators can give your applications a major performance boost. They're also called op-code caches and are generally a PHP extension, which caches compiled bytecode. Therefore the PHP code can be parsed and compiled once instead of for each request. Popular implementations are the **Alternative PHP Cache** (**APC**), eAccelerator, or XCache. This is an easy enhancement you shouldn't miss!

Creating our own short code

It's time to get started on the second part of this chapter: Short codes.

Think of short codes as placeholders inside the CMS' input fields. They are automatically replaced when the page is loaded. Instead of forcing content authors to add some complex HTML, you can do the heavy lifting for them. Let's build our own short code, as this will explain it best.

Time for action – how to find us

We want to add a map with the location of the bar—it must be easy for customers to find us! However we don't want to put the map in the sidebar (or footer), but right in the middle of `$Content` in the template.

If a widget satisfies your needs, go to the SilverStripe website where you can find readily available solutions for maps.

Let's create our own short code.

1. We'll need to register our short code in `mysite/_config.php`:

```
ShortcodeParser::get()->register(
  'GMap',
  array('Page', 'GMapHandler')
);
```

2. In `mysite/code/Page.php` add the following method to the Model:

```
public static function GMapHandler($attributes, $content = null,
$parser = null){

  if(!isset($attributes['location']) || !(strpos(
    $attributes['location'],
    'http://maps.google.com/maps'
  ) === 0)){
    return;
  }

  if(isset($attributes['width']) &&
      ctype_digit($attributes['width'])){
    $width = $attributes['width'];
  } else {
    $width = 700;
  }
  if(isset($attributes['height']) &&
      ctype_digit($attributes['height'])){
    $height = $attributes['height'];
  } else {
    $height = 530;
  }

  $customise = array();
  $customise['Location'] = $attributes['location'] .
      '&amp;output=embed';
  $customise['Width'] = $width;
  $customise['Height'] = $height;

  $template = new SSViewer('GMap');
  return $template->process(new ArrayData($customise));
}
```

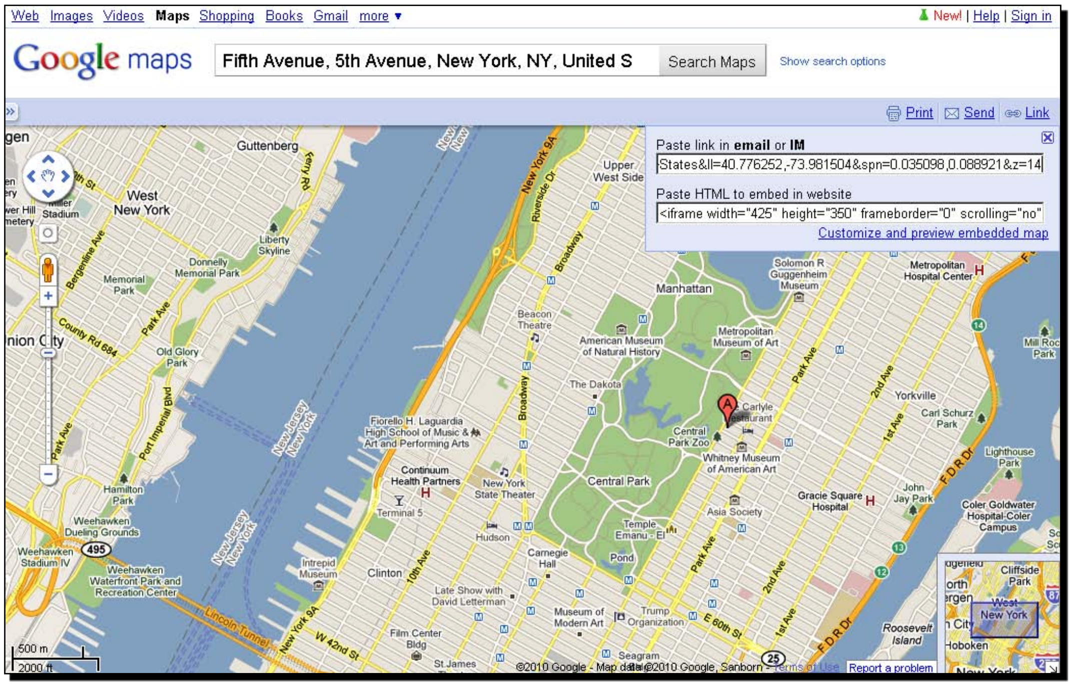
3. Create the file `mysite/templates/Includes/GMap.ss`—you'll need to create two new folders for that:

```
<iframe width="$Width" height="$Height" src="$Location"></iframe>
```

4. Rebuild your site.

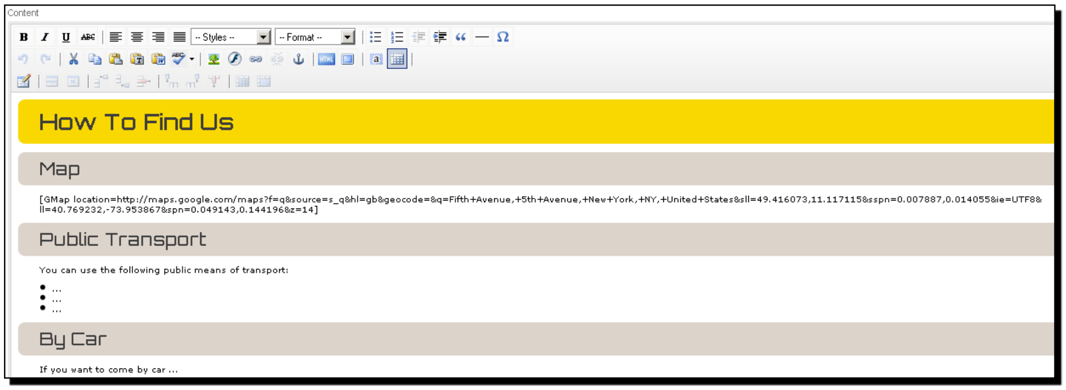
5. Go to `http://maps.google.com` and search for an address. In our example we went with New York's Fifth Avenue. Quite a nice spot, don't you think?

   Now copy the **Paste link in email or IM**.


Web Images Videos Maps Shopping Books Gmail more ▾
New! | Help | Sign in
Google maps
Fifth Avenue, 5th Avenue, New York, NY, United S
Search Maps
Show search options
Print Send Link
Paste link in email or IM
States&ll=40.776252,-73.981504&spn=0.035098,0.088921&z=14
Paste HTML to embed in website
<iframe width="425" height="350" frameborder="0" scrolling="no"
Customize and preview embedded map
Manhattan
Central Park
West New York
©2010 Google - Map data ©2010 Google, Sanborn - Terms of Use
Report a problem

6. Log into `/admin` and create a new location page for your site, just a regular content page is fine. Add the short code anywhere in the content area of the rich-text editor. You can position the map wherever you want in the flow of the page's content, giving control to the content author. You don't need to type the URL, simply copy it. The following is an example of our short code:

```
[GMap location=http://maps.google.com/maps?f=q&source=s_q&hl=gb&ge
ocode=&q=Fifth+Avenue,+5th+Avenue,+New+York,+NY,+United+States&sll
=49.416073,11.117115&sspn=0.007887,0.014055&ie=UTF8&ll=40.769232,-
73.953867&spn=0.049143,0.144196&z=14]
```



Content

How To Find Us

Map

```
[GMap location=http://maps.google.com/maps?f=q&source=s_q&hl=gb&geocode=&q=Fifth+Avenue,+5th+Avenue,+New+York,+NY,+United+States&sll=49.416073,11.117115&sspn=0.007887,0.014055&ie=UTF8&ll=40.769232,-73.953867&spn=0.049143,0.144196&z=14]
```

Public Transport

You can use the following public means of transport:

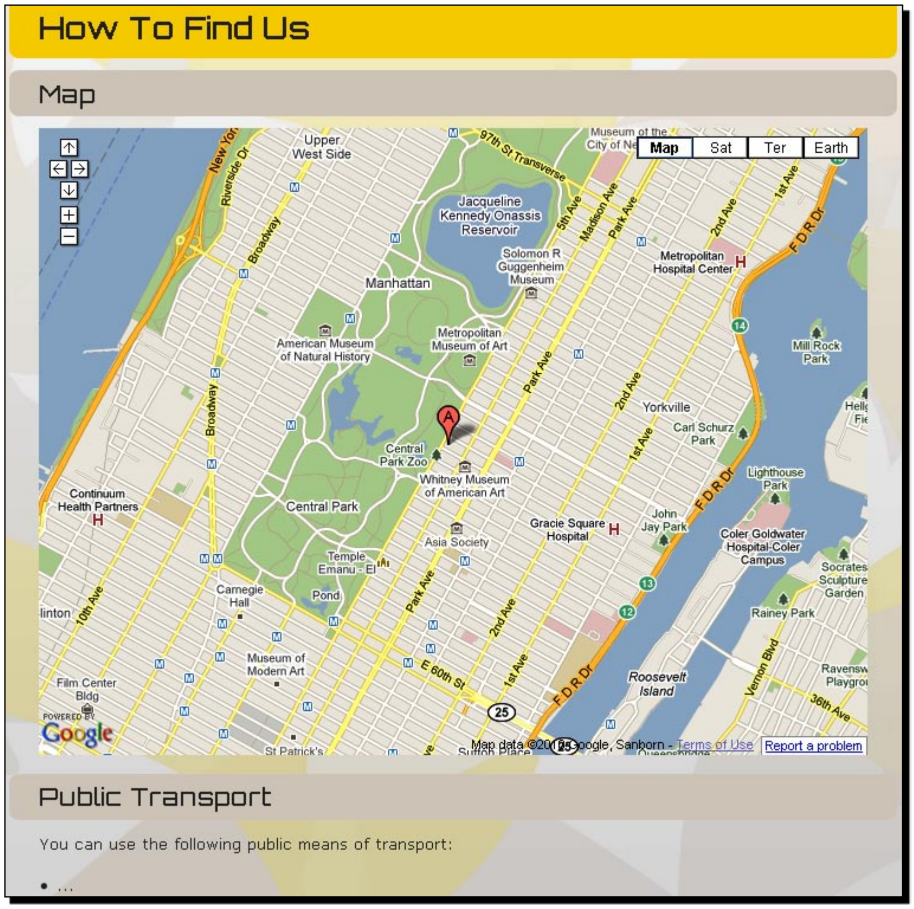
- ...
- ...
- ...

By Car

If you want to come by car ...

7. **Save and Publish** the page.

8. Go to the frontend and reload the page where you've added the short code.



How To Find Us

Map

Public Transport

You can use the following public means of transport:

- ...

9. All done!

> We've built our short code on Google Maps as it's widely used and free of charge, but you could use any other service you want. We're not trying to promote any specific provider.

What just happened?

Now that you've seen how it works, let's go over the code and see what actually happens.

What do we want to achieve?

The basic idea is to include the map from Google in an `iframe` on our page. For doing that, we create our short code which is then replaced with the external content by the CMS.

As we've already seen, short codes are always enclosed in squared brackets. HTML markup uses angle brackets but is otherwise very similar. With both, you can use a single tag or a starting and ending tag, optionally with content in between and you can add zero or multiple attributes. So the following examples are all valid:

- [Tag]
- [Tag]Content in between[/Tag]
- [Tag firstattribute=value secondattribute="quoted value"]

Note the conventions:

- Tag is always replaced by the template method of the same name, so they should be in UpperCamelCase
- Attribute names will always be lowercased in the Model, so keep them like this too
- Template placeholders should be Uppercased
- Attribute values can be quoted (if they contain a space, they must be)

So if this could be done with plain HTML, why are we creating short codes? On the one hand it's easier for content authors and usability should always be a design goal. On the other hand it's a security issue. You'll want to be able to control which external pages can be embedded into your site.

Configuration

In the configuration we register our short code, so the system can replace it:

```
ShortcodeParser::get()->register()
```

The first argument is the name of the tag we want to use in the content field. As we want to use `[GMap target=...]` it's `GMap`. The second argument is an array where we define that this short code maps to the `GMapHandler` method in the `Page` class. The method name can be freely chosen, but attaching handler to the short code is a logical choice which will be easily recognizable.

Model

So in the `Page` class we have our `GMapHandler` method. Public because it's referenced outside its own class, static because it's not bound to a specific object and only depends on the input—but you already know that.

General composition

Before going into the specific details of our example, let's cover how this works in general terms.

The first argument contains a named array with all attributes. The second one contains the optional content between tags, and the third one contains the optional parser to use for this method. The names of the attributes are of course up to you.

In the end, we return which 'include' template to use in order to render our short code, including any variables it needs.

`$template = new SSViewer('GMap');` defines the template to use. It will first look for a file `mysite/templates/Includes/GMap.ss`, if that isn't found it will try `templates/Includes/GMap.ss` in your theme's directory.

We return this template and add (`->process()`) a named array of the variables the template uses. SilverStripe has its own wrapper `ArrayData` for the inclusion of variables, so the template can handle the array data.

Example-specific code

The second and third arguments are not used in our example, so we can ignore them.

We first check that the mandatory attribute location is present and that it is indeed a Google Maps address. The second check is to ensure careless or malicious content editors can't include unwanted content in our page. It's a good idea to check that. If there is anything wrong, we simply return nothing, thereby replacing the short code with nothing.

We then check whether the optional attributes `height` and `width` are used, and if they are a valid number. If they fail the validation, we'll use default values.

For the template, provide the appropriate width and height, and also make sure that the Google Map is used in the embedded mode. It's better to do that in the code rather than make content editors fumble with the link—either forgetting or breaking it.

So valid short codes for this method could be (replacing the Google Map URL with three dots):

- [GMap location=...]
- [GMap location="..."]
- [GMap location=... width=200]
- [GMap location=... width=200 height=200]
- [GMap location=...]I'm ignored[/GMap]

Template

First off, why didn't we put the include in the `templates/` directory—didn't we say that all View related code should be saved there?

Well, it depends. It's of course a valid approach and you can safely copy the include there and remove the empty folders in `mysite/`. However, our code doesn't really contain any layout-specific parts. It's just the basic structure which we would need to copy to all themes. Copying code to different locations should always raise
a warning flag—that's anti-DRY. It's kind of a trade-off. As a rule of thumb:

- If it's just structural markup that's the same for all themes, keep a single copy in `mysite/`
- If it may change depending on the layout or if you actually need varying markup, put it in your themes

As you can see, we can mix both approaches as needed.

Otherwise there's nothing special happening. We use the three named array elements of the `$customise` variable from the Model in our `iframe`: `$Width`, `$Height`, `$Location`.

And it even validates, when using the HTML5 doctype at least. Note that you can have validation issues when using an `iframe` with XHTML.

Common pitfalls

You obviously need all the parts we've just covered. Besides naming and referencing the files and methods correctly, don't forget to rebuild your site.

If the short code is displayed untouched in the frontend, you either didn't register the right code in the configuration or you didn't rebuild the site.

Once you've created the short code and it's working as expected, think about how to document it. While it's in the code, that's invisible to content authors. Spend some time writing all short codes and their options down and providing that information to everyone involved. Otherwise there's a pretty good chance of features being forgotten in the long run...

Go forth and build your own

Now that we know how to create a custom short code for maps, think about other use cases for it.

SilverStripe itself uses short codes for internal links: `[link id=1]Link to the page with ID 1[/link]`. So if you change a page's URL, the link is automagically updated. Including videos could be another common example, but there are many other clever things you can possibly achieve.

Pop quiz – too many options?

So far we've discussed four methods for extending the CMS:

- Adding custom fields to a page
- Adding custom fields to the site configuration
- Creating a widget or using an existing one
- Creating your own short code

Take a look at the following requirements. The four approaches have pretty distinct use cases and you'll make your life needlessly hard if you try to achieve something with the wrong approach. Which one(s) fit the requirements best?

1. You want to add the same advertisement on all of your pages for a special offer.
2. You want to display the offer from the previous point, but only on some selected pages.
3. You want to display your special offer right in your `$Content`.
4. You want to display different offers on various pages. Some pages shouldn't contain any advertisement.

Summary

In this chapter, we made good use of our knowledge from the previous chapters to add some nifty features to our page. Besides implementing our own widget and short code, we also explored when to use which approach: Short codes are added into CMS fields while widgets are placed through a dedicated placeholder in the template. We've also explored caching, making our page perform a lot better and using fewer resources. While moving along we also picked up the concept of text parsers.

Using widgets, we have integrated our Facebook status messages into the website, so we don't need to manually copy the content any more.

Using short codes we've added the feature to include a map anywhere in the content area of our page.

While this has already helped us getting pretty far, there are limits. For example adding an image gallery, something that we definitely need for our bar, is not possible with the tools we've covered so far. But that's one of the tasks we'll tackle in the next chapter.

7

Advancing Further with Inheritance and Modules

In the previous chapter we began to customize our site with widgets and short codes. In this chapter we'll add some more advanced customizations, building on two very popular (third party) modules. Specifically we'll add support for images in our system, both in the templates and managing them in the CMS.

From a technical point of view we shall:

- Get to know the DataObjectManager module
- Extend it with the Uploadify module
- Rework our rotating images on the intro page
- Add our own image galleries
- Build our own module

So let's get on with it!

Enhancing the intro page

What elements of the page can be improved? Well, the images in our intro page are hard-coded into the template. Every time a content editor wants to change the order, or add or remove an image, he needs someone with technical skills to do that for him. Let's fix that.

Time for action – adding the required modules

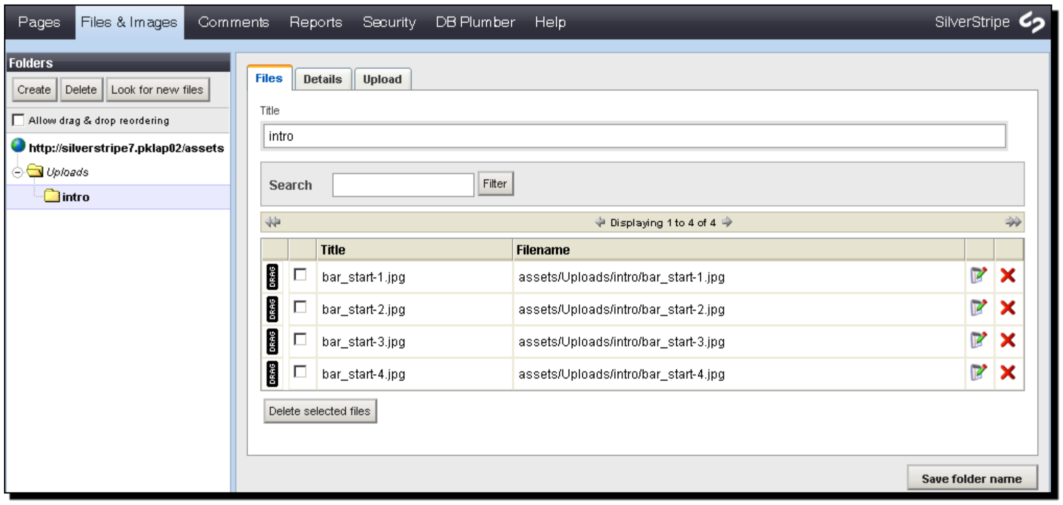
This part will introduce some great SilverStripe extensions, which add even more power to the system. This allows us to make the CMS very user-friendly for content authors, but it also makes things a little more complicated for developers. However, the effort is well spent, as you'll see. Let's dive right in:

1. Get the latest version of the DataObjectManager at `http://www.silverstripe.org/dataobjectmanager-module/`. It's a very common module and is often abbreviated as DOM. Don't mix it up with the Document Object Model, also abbreviated as DOM, for interacting with HTML and XML documents. Both are commonly used in SilverStripe, so take care to differentiate the two.
2. Make sure you include the folder in your root directory and that it is called `dataobject_manager/`.
3. Also get Uploadify at `http://www.silverstripe.org/uploadify-module/`.
4. Make sure that the folder is called `uploadify/`.

[Both modules are very helpful extensions to the SilverStripe core and are actively maintained by Aaron Carlino, aka Uncle Cheese, a member of the open source community who is not associated with SilverStripe Inc.]

5. Rebuild the database.
6. Browse to `/admin/assets/`.
7. The **Files & Images** tab before the addition of the new modules looks like the following image:

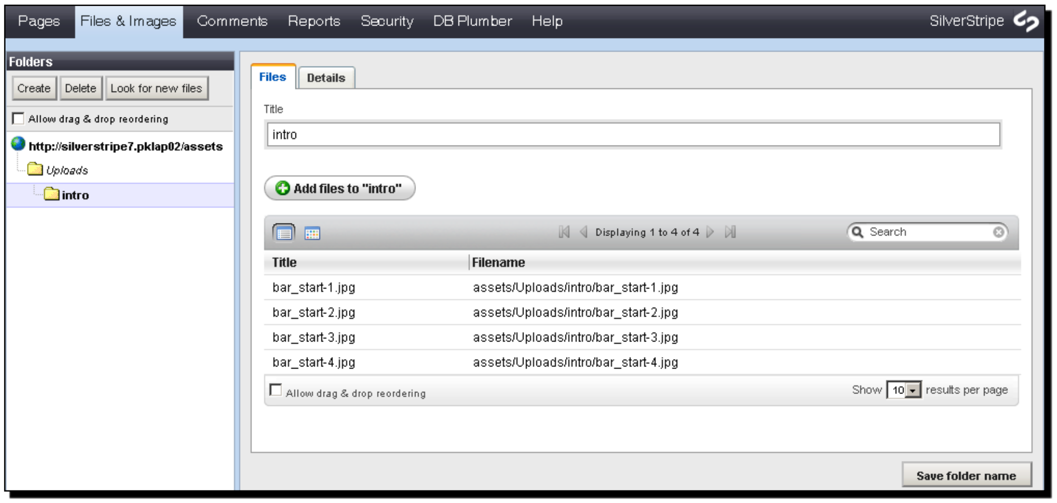
[If you've added the files in the file system, you need to click the button **Look for new files** in order to synchronize them with SilverStripe's database. If you add files in the CMS directly, there's no need for this step.]



Title: intro

Title	Filename
bar_start-1.jpg	assets/Uploads/intro/bar_start-1.jpg
bar_start-2.jpg	assets/Uploads/intro/bar_start-2.jpg
bar_start-3.jpg	assets/Uploads/intro/bar_start-3.jpg
bar_start-4.jpg	assets/Uploads/intro/bar_start-4.jpg

8. After installing the modules, it looks like this:



Title: intro

Title	Filename
bar_start-1.jpg	assets/Uploads/intro/bar_start-1.jpg
bar_start-2.jpg	assets/Uploads/intro/bar_start-2.jpg
bar_start-3.jpg	assets/Uploads/intro/bar_start-3.jpg
bar_start-4.jpg	assets/Uploads/intro/bar_start-4.jpg

9. While the asset manager's new look is quite nifty and also provides a file preview, it omits one feature of the original implementation: moving files. Depending on your requirements you might absolutely need this function while other developers might not even know it existed.

There is a simple fix for switching back to the original view. Add the following line to `mysite/_config.php`. This changes back the asset manager's look and functionality to the default, but we'll still be able to use other DOM features:

```
DataObjectManager::allow_assets_override(false);
```

What just happened?

The DataObjectManager module's tools enable easy and robust handling of DataObjects related to a page. Additionally, it's easier to upload and manage files and images than when using the out-of-the-box SilverStripe CMS installation.

For working with files, we're also using the Uploadify module—it handles the upload and inclusion process. The module is a SilverStripe wrapper for the excellent JavaScript library Uploadify (`http://www.uploadify.com`). Although SilverStripe's default installation can also handle images (upload multiple files, rename, move, and delete), Uploadify integrates better into DOM and makes it easier to add and import images in a single step.

To summarize, we included these two modules, but reverted the changes to **Files & Images**.

Time for action – extending the backend functionality

Now that we've added and configured the required modules, we can start enhancing our code. We'll start with the backend part concerning the CMS:

1. Create a new class `mysite/code/CustomImage.php`, which is the container for our individual images and links them to specific pages:

```
<?php

class CustomImage extends DataObject {

  public static $has_one = array (
    'BaseImage' => 'Image',
    'BelongToPage' => 'Page',
  );

  public function getCMSFields(){
    return new FieldSet(
      new FileIFrameField('BaseImage')
    );
  }

}
```

2. Add the following line to `mysite/_config.php` to allow easy reordering of images:

   ```
   SortableDataObject::add_sortable_class('CustomImage');
   ```

3. Connect any number of images to your intro page by adding the following code to `mysite/code/IntroPage.php`:

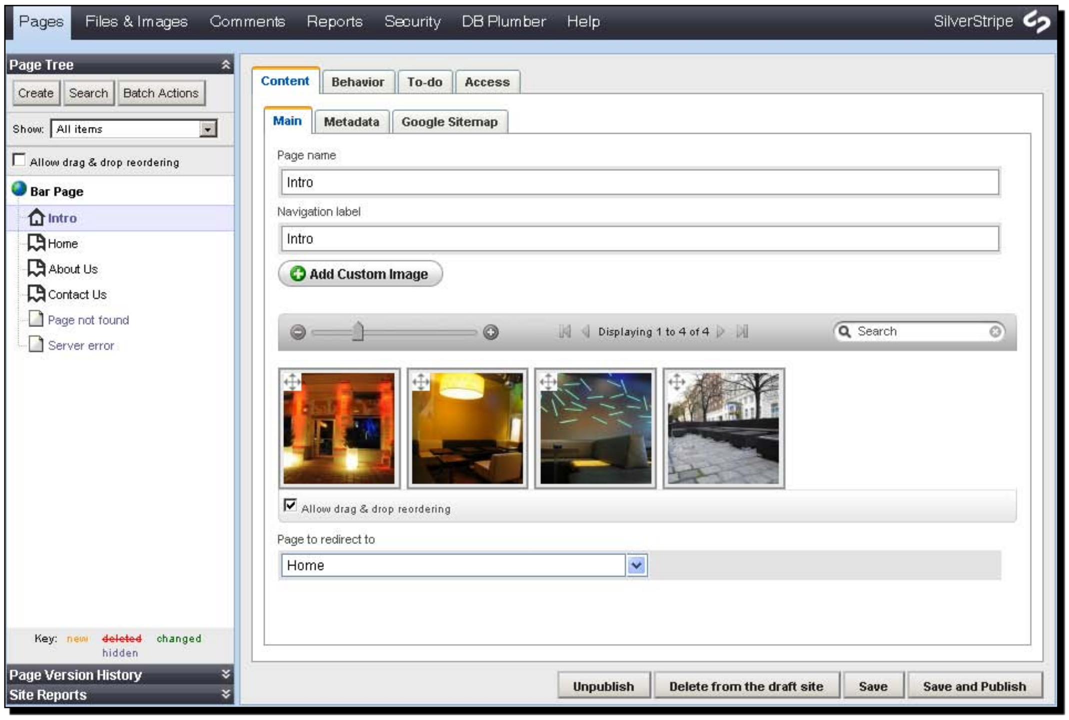
   ```
   public static $has_many = array(
     'CustomImages' => 'CustomImage',
   );
   ```

4. In the same file, extend your `getCMSFields()` method with the following code (new lines are highlighted) to enable editing of images in the CMS:

   ```
   public function getCMSFields(){
     $images = new ImageDataObjectManager(
       $this,
       'CustomImages',
       'CustomImage',
       'BaseImage'
     );

     $fields = parent::getCMSFields();
     $fields->addFieldToTab('Root.Content.Main', $images);
     $fields->addFieldToTab('Root.Content.Main',
     new TreeDropdownField('PageRedirectID', 'Page to redirect to',
                           'SiteTree'));
     $fields->removeFieldFromTab('Root.Content.Main', 'Content');
     $fields->removeFieldFromTab('Root.Content', 'Widgets');
     return $fields;
   }
   ```

5. Rebuild the database again.

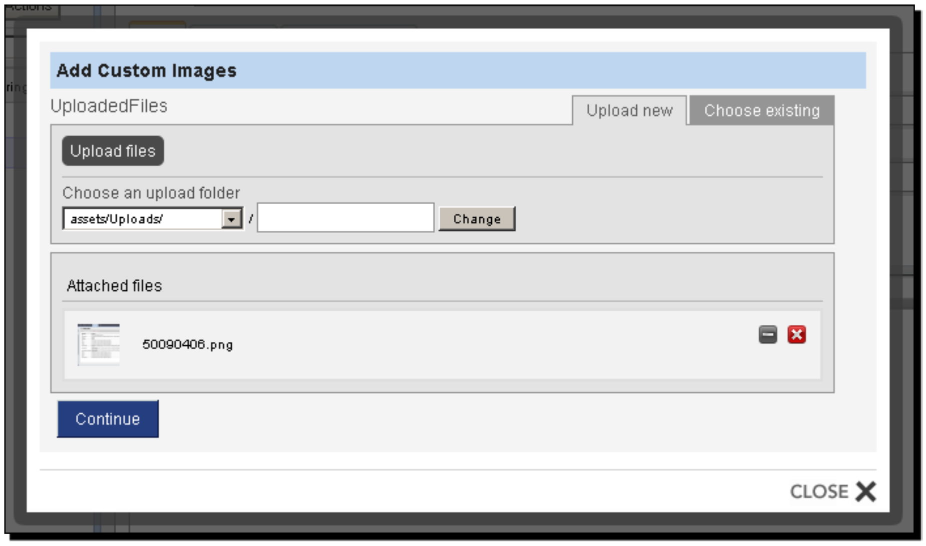
6. The intro page type in the CMS will now look like this— the hard-coded images from the template have already been (manually) added to illustrate the functionality:



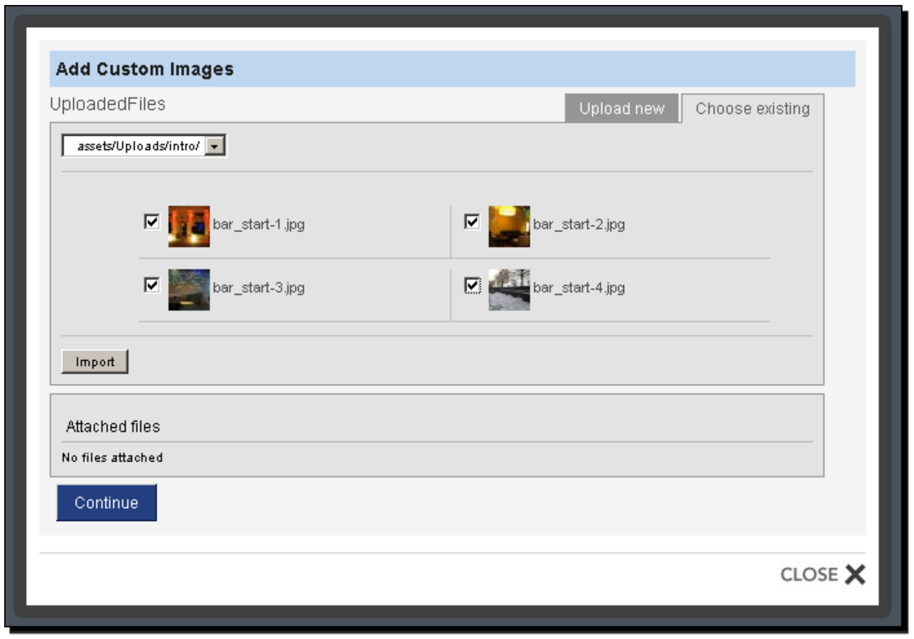
7. Now it's your turn to add some images. In the CMS click on the **Add Custom Image** button.

8. This opens up a new window. Here you can either import previously uploaded pictures (**Choose existing**) or add new ones (**Upload new**).

9. If you upload new images, you can select the target folder inside `assets/Uploads/`. Having all intro images in a single folder, which you can easily add in the asset manager, helps you keep everything organized. We're using `assets/Uploads/intro/` for our target folder. Add images by clicking on the **Upload files** button, select the files, and click on the **Continue** button. It's a very straightforward process.



10. If you want to reuse existing images, select their folder. Wait for the view to automatically load the pictures, select the ones you want to use, and finally **Import** them. It's just as easy as adding new images.



What just happened?

In order to appreciate what we've done, it's now time to take a detailed look at some of the fundamentals of Sapphire. Once you've understood these concepts, you can build powerful features, like our CMS-based intro page.

Introducing the DataObject class

Our `CustomImage` class is a container for our images, adding information to the bare picture. We connect an individual image (called `BaseImage` in our example) with a page (`BelongToPage`) where it should be displayed. Using the has-one relationship we define that an object based on our `CustomImage` class contains one image and belongs to one page. Image and Page are predefined SilverStripe data-types. Their usage is pretty self-explanatory. We could have used `IntroPage` instead of `Page`, but let's keep it generic at the moment—later in this chapter you'll see why this is useful.

For adding the image to a page, we rely on the already well-known `getCMSFields()` method. The field-type `FileIFrameField()` provides the basis for adding files in the CMS.

But why are we suddenly extending DataObject and not Page anymore? There are basically three data-types we might want to use (plus any children we've created and need to extend) for adding functionality to our site. These data-types are:

SiteTree

Every class that inherits from SiteTree is available as a page-type in the CMS. The edit form in the backend, URL, versioning, and more are also added thanks to this base class.

In `mysite/code/Page.php` we extended SiteTree, laying the foundation for all other pages.

Base your class upon SiteTree (directly or indirectly) if you want to create a (stand-alone) page. Our intro and content pages are good examples for this use case.

DataObject

Anything you want to store in the database should be based on DataObject, providing the basics for saving and loading information. Actually, SiteTree inherits from it, adding the features we've just discussed.

In case you don't need the SiteTree's features, get rid of them by inheriting from DataObject directly—specifically when some information doesn't require a dedicated page, its own URL, and so on. The `CustomImage` class is a good example for that as it's only used as a part of a page.

DataObjectDecorator

We've already used the DataObjectDecorator class, as you may remember. It was used in the `CustomSiteConfig` class for adding functionality to the already existing SiteConfig. In other systems this is often called a **Mixin**.

If you have a subclass of DataObject, you can easily extend it (add methods and properties) with the help of this class. But why would you do that instead of simply subclassing? Well, there are some classes the system uses by default (SiteConfig for example) which we could subclass, and subsequently change any reference to our new class. But this would mean changing the core code, making it harder to maintain and upgrade. A better approach is to extend a class and hook our new functionality into it, just as we did with `CustomSiteConfig`.

Making DataObjects sortable

After adding images, we might want to reorder them. The following snippet allows content authors to choose the order by drag and drop in the CMS:

```
SortableDataObject::add_sortable_class('CustomImage');
```

The name SortableDataObject highlights what is happening here: we want to have a sortable DataObject of our newly-created `CustomImage` class.

After setting this option you need to rebuild your database as a new column must be added to the table, storing the order you defined.

Adding our DataObject to the intro page

As always there are two steps in this process: First we need to add the new field to our database. Then we need to make it accessible in the CMS so that we can actually make use of it.

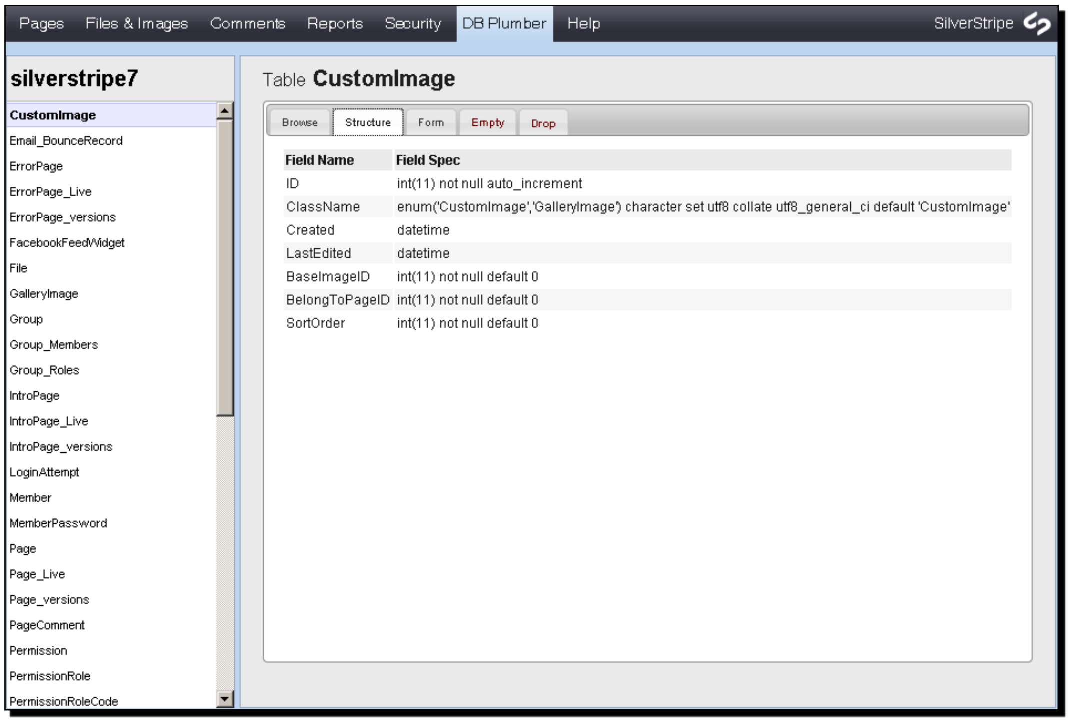
Storing required information in the database

We want to allow the intro page to have any number of images, including none at all. To do this, we can use the `$has_many` relationship in the `IntroPage` class in the `mysite/code/IntroPage.php` file:

```
public static $has_many = array(
  'CustomImages' => 'CustomImage',
);
```

We're naming our relationship `CustomImages` and it's of the type `CustomImage`—the class we've just created. It's not required to use this pattern of pluralizing the class name as a relationship name, but it's a common and helpful convention. Unless you have a good reason, stick to it in `$has_many` relationships.

This `$has_many` directly interacts with the `'BelongToPage' => 'Page'` of the `CustomImage` instance's `$has_one`. You need to define this kind of relationship on both sides for it to work. On the database level, everything is stored inside the `CustomImage` table, which you can see best in DBPlumber:



Field Name	Field Spec
ID	int(11) not null auto_increment
ClassName	enum('CustomImage','GalleryImage') character set utf8 collate utf8_general_ci default 'CustomImage'
Created	datetime
LastEdited	datetime
BaseImageID	int(11) not null default 0
BelongToPageID	int(11) not null default 0
SortOrder	int(11) not null default 0

As you can see, everything is neatly stored here: which image should be used, to which page it belongs, and the sort order.

We've not added a `$has_many` in the base image class—so that one image can be used in multiple `CustomImage` objects. But that's not necessary; SilverStripe handles that automagically for us.

Providing a field in the CMS

In order to allow content authors to manage images, we only need to add one field, which allows us to add any number of images. The easy part is the following line, which goes into the `getCMSFields()` method:

```
$fields->addFieldToTab('Root.Content.Main', $images);
```

As the field is a bit more complex we moved it out of `addFieldToTab()`—you can do that wherever you see the need for it.

Let's look at the rest of our newly-added code:

1. `$images = new ImageDataObjectManager(`

   As we're working with an image we rely on `ImageDataObjectManager()`. If you want to handle any file use `FileDataObjectManager()`.
2. `$this,`

   The first argument defines the Controller in which the field should be available. The easiest and most portable way of doing this is simply referencing it with `$this`—unless you're in a decorator, in which case it's `$this->owner`.
3. `'CustomImages',`

   The name of the relationship we want to use in this field, defined in the `$has_many`.
4. `'CustomImage',`

   The base class of the field.
5. `'BaseImage',`

   The name of the file field actually used, taken from `'BaseImage' => 'Image'` in the `CustomImage` class.

Optional arguments

All arguments, except for the first one, are optional. `CustomImages` for instance would be automatically resolved as it's the only `has_many` relation on the page. But for clarity we're adding the optional arguments.

Quite a few options here, but it provides a very powerful field in the CMS. That's the backend part, so let's move on.

Time for action – integrating our changes into the frontend

Now that we're done adding images via the CMS, it's time to update our templates so that they appear to visitors:

1. Open `themes/bar/templates/Layout/IntroPage.ss` and replace the `<figure>` (HTML5 tag, in XHTML it would probably be a `<div>`) part with:

```
<figure id="rotator">
  <ul>
    <% if CustomImages %>
      <% control CustomImages %>
```

```
                <li<% if First %> class="show"<% end_if %>>
                  <a href="$Top.PageRedirect.URLSegment">
                    <% control BaseImage %>
                      <% control SetSize(720, 478) %>
                      <img src="$URL" alt="Intro image number $Pos"
                         class="rounded transparent-nonie shadow"/>
                      <% end_control %>
                    <% end_control %>
                  </a>
                </li>
              <% end_control %>
            <% end_if %>
          </ul>
        </figure>
```

2. Reload the intro page. It should look just like before in the frontend, but now we can add and remove images as well as reorder them via drag and drop in the CMS.

What just happened?

The template is the final piece missing in our intro page puzzle. We'll also use it to introduce SilverStripe's image manipulation capabilities.

Working with images

What we want to achieve in the template is the images we added in the backend being automatically displayed in the frontend. The logic for this is quite complex:

- The outer `<figure>` and `<ul>` are the same as before, simply providing the correct markup.
- First we need to check if any images have been added to the page: `<% if CustomImages %>`. Trying to loop over non-existing entries might lead to errors. If images are found, we issue a `control` statement for looping over all available entries.
- The next line (`<li<% if First %> class="show"<% end_if %>>`) ensures that we initialize only the first of our images by applying the CSS class `show` to it. The other images rely on the same structure but without the class attribute.

- We then link every image to the start page we defined in the CMS. `PageRedirect.URLSegment` we already know about but what about the `$Top`? It's needed because we're currently in the `CustomImage` context (due to the `<% control CustomImages %>`). Inside our `CustomImage` control block there is no `$PageRedirect` placeholder available. It only exists in the global scope. Therefore, we need to "break out" of this context to be able to access it.

`$Top` won't get you out of a single context (in case multiple ones are wrapped together), but out of all of them—directly back to the global scope.

- After all of these steps, we've now finally reached the layer of our image itself. Drilling down to its details is done with `<% control BaseImage %>`. This isn't a loop, as there is just a single `BaseImage` in each iteration.
- `<% control SetSize(720, 478) %>` ensures that our image has the right dimensions—720 pixels wide and 478 high. So our content editors don't need to start Photoshop and manually resize images before adding them in the CMS. SilverStripe handles that for us and we won't accidentally break the layout if an image doesn't have the right dimensions. Actually, images are resampled and not just sized down.
- Finally we add our image. Due to the `<% control BaseImage %>`, `$URL` is actually `$BaseImage.URL`—the URL of our resized image—just what we need for `<img src=""/>`.

All done! Hopefully that breakdown was useful. While we're at it, let's take another look at the resizing of images.

Image functions

SilverStripe provides several functions to work on images. You can use them in both the Controller and the View. Depending on your use case you can decide when to use which approach; in our case the template is a good option.

Resized images are automatically cached for better performance. In the folder of the resized image a subfolder `_resampled/` is automagically created containing any resampled versions. In order to get rid of them or regenerate them, you can either delete them manually or append `/images/flush` to the current URL (as already covered that in *Chapter 3*, *Getting "Control" of Your System*).

Images in the Controller

You can issue the following method calls on an image, an example would look like this:

```
$image->croppedResize(80, 80);
```

Method	Description
`croppedResize(width, height)`	The image is cropped to the given dimensions, starting in the center. Remaining parts are cut off. If the image is too small it is padded.
`getHeight()`	Gets the height of the image.
`getOrientation()`	Returns one of the three constants, depending on the greater dimension: `ORIENTATION_SQUARE`, `ORIENTATION_PORTRAIT`, or `ORIENTATION_LANDSCAPE`
`getWidth()`	Gets the width of the image.
`greyscale(r,g,b)`	Turns an image from color to gray. Using the arguments you can fine tune how the primary colors are shifted to gray.
`paddedResize(width, height)`	Resizes the image to the given dimensions. If the image is too small it is padded.
`resize(width, height)`	An image is resized to the given width and height.
`resizeByHeight(height)`	Defines the height, scaling the image up or down. The proportions are kept intact.
`resizeByWidth(width)`	Same as the previous one, but for the image's width.
`resizeRatio(width, height)`	Same as resize but keeps the ratio of the original image.

Images in the View

In the template you have nearly the same functionality, only the names are slightly different. Calling each function can be done either inside a control structure or directly via `$Image.URL` for example.

Method	Description
`CroppedImage(width, height)`	Scales an image to the given dimensions and applies cropping if required.
`Filename`	Gets the filename of the current image. For example: `assets/Uploads/intro/_resampled/SetSize720478-bar_start-1.jpg`
`Height`	Similar to `getHeight()` in the Controller.
`Orientation`	Returns the orientation as a number. A square image has the value of `0`, a portrait `1`, and landscape `2`.

Method	Description
`PaddedImage(width, height)`	The image is scaled accordingly but not cropped. Unused space is padded.
`SetHeight(height)`	Similar to `resizeByHeight(height)`.
`SetRatioSize(width, height)`	The image is scaled according to the bigger value with respect to the original dimensions.
`SetSize(width, height)`	Sets the size to the given values. The ratio is kept intact, but if the image is not square it is padded.
`SetWidth(width)`	Similar to `resizeByWidth(width)`.
`Width`	Similar to `getWidth()`.
`URL`	Gets the URL of the current image. For example: `/assets/Uploads/intro/_resampled/SetSize720478-bar_start-1.jpg`

Pop quiz – SiteTree, DataObject, or DataObjectDecorator

Selecting the right data-type is crucial for getting the right mix of features without making your life unnecessarily difficult.

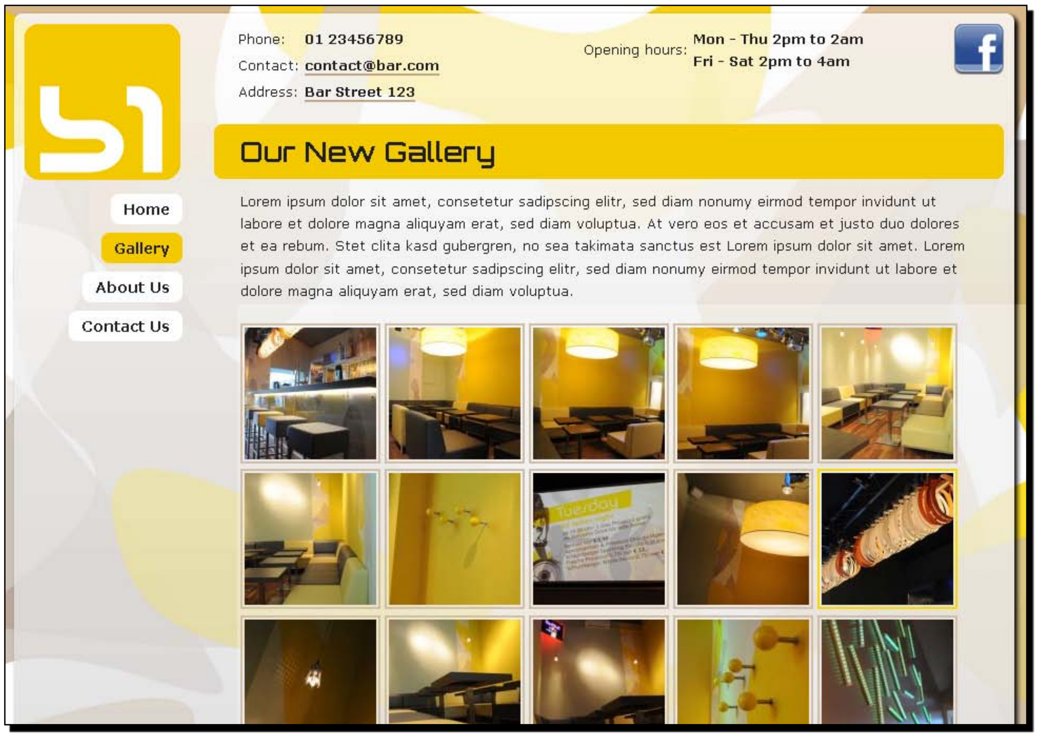
Assume you want to create your own image gallery. You want to easily add images in the CMS. Thumbnails should be displayed in the frontend. If one thumbnail is clicked on, a bigger version should pop up. Which of our three base classes would you use for achieving this task?

- Just `SiteTree`: We want to create a new image gallery page, so that's it.
- Just `DataObject`: All we want to do is show some images so nothing else is needed.
- `SiteTree` and `DataObject`: We're only extending the intro page, the basis stays the same.
- `SiteTree` and `DataObjectDecorator`: We need a page, so `SiteTree` is necessary. Additionally, we want to inject some information into the base `Image` class via the `DataObjectDecorator`.

If you're unsure what the correct answer is, don't worry. We'll start building our own image gallery right away.

Image galleries

The author of DOM and Uploadify has already released an image gallery: `http://www.silverstripe.org/imagegallery-module/`. However, as this module can't be easily used for the intro page and we will need to change some parts anyway, we'll create our own. And we'll learn a lot more that way, so let's get started. The final result should look like this:



Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

While there are some very good extensions available, there are far fewer than for other popular CMS, because SilverStripe focuses on easily writing custom parts yourself rather than relying on too much pre-built code. At first, this looks like a disadvantage, but if you have ever tried to find the perfect mix of extensions between thousands of modules, only to find that it doesn't exist, you should appreciate it. It saves you a lot of trouble from outdated, incompatible, insecure, and bloated extensions.

Time for action – creating our own image gallery in the CMS

We could have started from scratch, but our aim here is to reuse as much of the intro page as possible. That's what object-oriented programming is all about after all.

1. Create a new file `mysite/code/GalleryImage.php`:

```
<?php

class GalleryImage extends CustomImage {

  static $db = array(
    'Title' => 'Text',
  );

  public function getCMSFields(){
    return new FieldSet(
      new TextField('Title'),
      new FileIFrameField('BaseImage')
    );
  }

  public function ResizedBaseImage(){
    $image = $this->BaseImage();
    if($image->getOrientation() == 2){
      return $image->SetWidth(720)->URL;
    } else {
      return $image->SetHeight(719)->URL;
    }
  }

}
```

2. Create a new file `mysite/code/GalleryPage.php`:

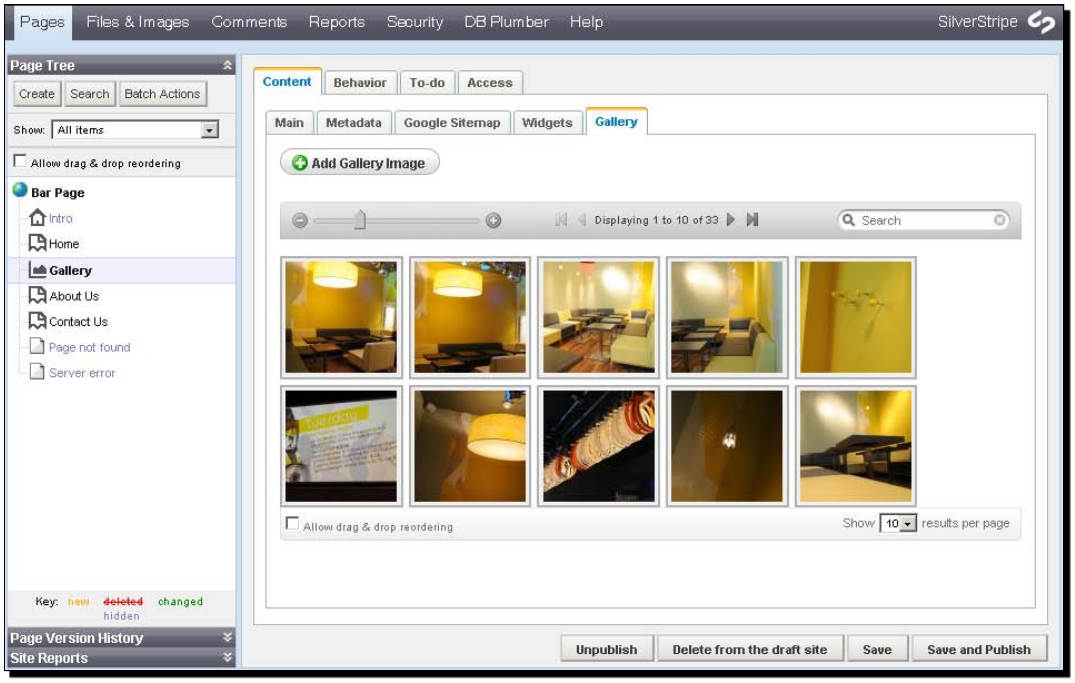
```
<?php

define('GALLERY_ICON_PATH', PROJECT_DIR . '/icons/gallery');

class GalleryPage extends Page {

  static $has_many = array(
    'GalleryImages' => 'GalleryImage',
  );
```

```
    public static $icon = GALLERY_ICON_PATH;

    public function getCMSFields(){
      $images = new ImageDataObjectManager(
        $this,
        'GalleryImages',
        'GalleryImage',
        'BaseImage',
        array(
          'Title' => 'Title',
        ),
        'getCMSFields'
      );
      $fields = parent::getCMSFields();
      $fields->addFieldToTab("Root.Content.Gallery", $images);
      return $fields;
    }

  }


  class GalleryPage_Controller extends Page_Controller {

    public static $allowed_actions = array();

    public function init(){
      parent::init();
      Requirements::css(
        PROJECT_DIR . '/thirdparty/colorbox/colorbox.css',
        'screen,projection'
      );
      Requirements::javascript(
        PROJECT_DIR . '/thirdparty/colorbox/jquery.colorbox-min.js'
      );
      $js =
  <<<JS
        $(document).ready(function(){
          $("a[rel='gallery']").colorbox();
        });
  JS;
      Requirements::customScript($js);
    }

  }
```

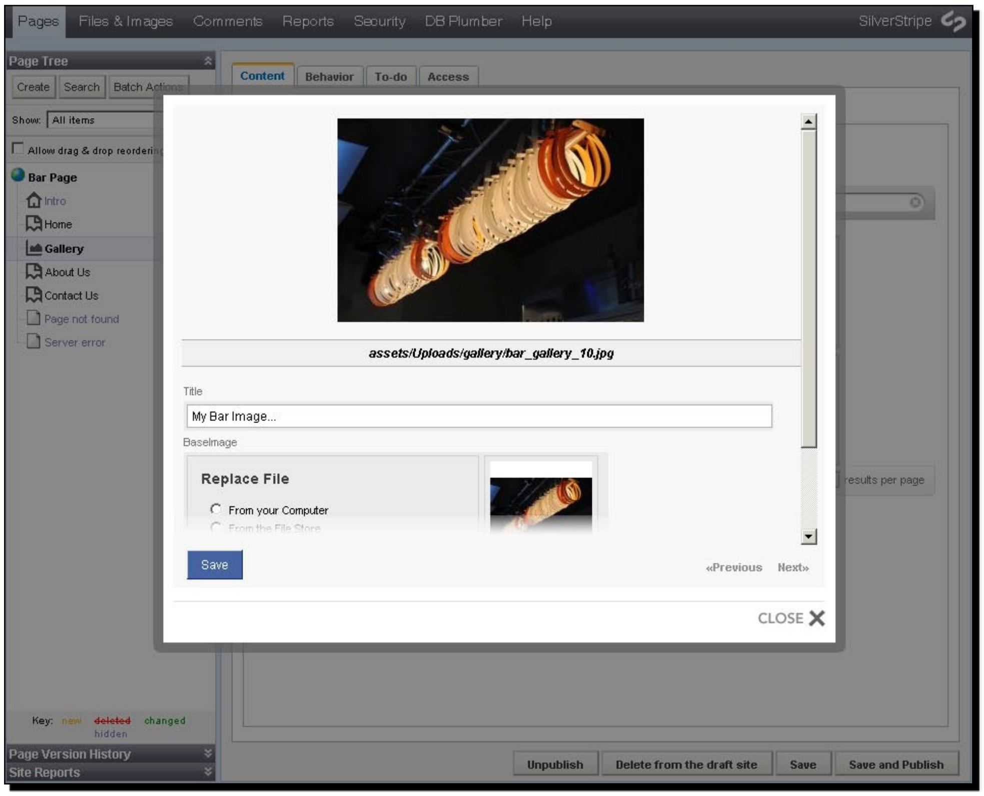
Adding custom JavaScript in the Controller should be avoided if possible. Generally you should create a `.js` file and link to that. But in this example we wanted to illustrate how JavaScript can be added using the heredoc notation (the `<<<JS / JS;` section). More on this later.

3. Add the icon for our page class.

4. After rebuilding the database, the final page in the CMS looks like the following image. Again some images have already been added.



5. Clicking on an image provides a similar popup to that seen on the intro page. This time, however, there's also a text field for adding a caption to the image:



What just happened?

This example takes our intro page even further:

A single gallery image

In the `GalleryImage` class we have reused our `CustomImage`. We inherit the `$has_one` relationships for tying images and pages together. Additionally we add a title to each image, and we overwrite the `getCMSFields()` function to make our title field usable.

Finally we've added the new method `ResizedBaseImage()`. In it we access the image contained in this object via `$this->BaseImage()`. That's the same name we're using to access the image in the `FileIFrameField`() method. Once we have the image, we check whether it's a landscape or a portrait / square. The longer side is then resampled to the given value (in pixels) and the URL of the resized file is returned. We'll make use of this function in the next step.

Image gallery container

The model `GalleryPage` is very similar to the one from the `IntroPage` class. The main difference is that we have added an additional optional field. It maps the added field `Title` from the `GalleryImage` class to the pop-up for editing the specific image. If we had more than one field, we'd include them in the array too.

> Our image gallery is sortable because this property has been inherited from the base class `CustomImage`.

In the Controller we use the `init()` function for setting up the required CSS and JavaScript. Note that we haven't added the referenced files yet; we'll do that in the next step. The only thing slightly different than in the previous snippets is the way in which the script is included.

> **Heredoc**
>
> This method is called **heredoc**, starting with `<<<JS` and using everything until `JS` as an input string. You can use any other name, not just `JS` as long as that name doesn't appear in the script itself.
>
> It's basically the same as double-quotes with the advantage that you don't need to escape any other double-quotes in the script. As this would have been the case in our code, we went with the heredoc notation.

Time for action – our image gallery in the frontend

Now that we've prepared the backend, let's take care of the frontend and let's not forget to add the two files we've already referenced.

1. Get ColorBox at `http://colorpowered.com/colorbox/`. It's a freely usable (MIT licensed) JavaScript library for image galleries.

> **Why ColorBox?**
>
> While there are many alternatives, like Fancybox, Lightbox, prettyPhoto, and some more, ColorBox fits our needs. It builds upon jQuery, is lightweight (only 10 KB compressed), unobtrusive, standards compliant, and remains feature rich. But if you prefer another solution, that's no problem—switching should be easy. All you need to change is the reference to the specific library in JavaScript.

2. Get the file `jquery.colorbox-min.js` from the download and place it in the `mysite/thirdparty/colorbox/` folder.

> The recommended place for storing third-party libraries is the folder `thirdparty/`. To avoid any collisions, each module should be saved into its own subfolder.

3. ColorBox provides five different styles by default. You can take a look at number one at `http://colorpowered.com/colorbox/core/example1/index.html`— replace `example1` with `example2` to `example5` to see the others. Copy the styling you like most to your `mysite/thirdparty/colorbox/` directory. In our example it's `example1/colorbox.css`.

4. Add the images accompanying the stylesheet to your `mysite/thirdparty/colorbox/images/` folder.

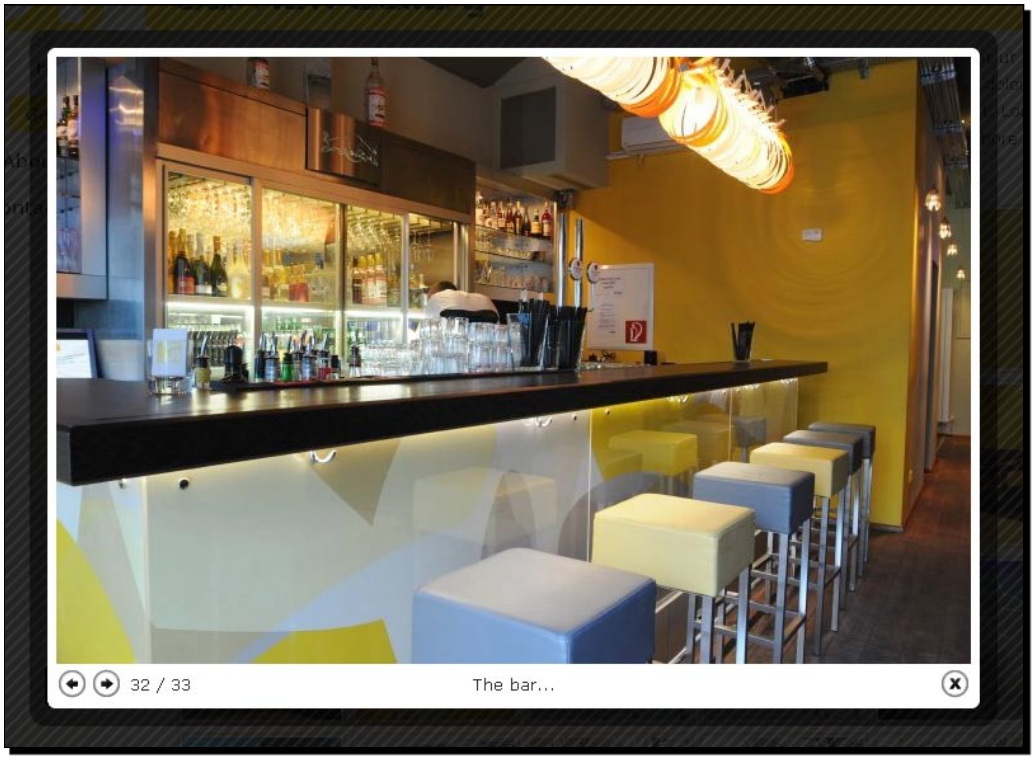
> If you follow this structure, you don't need to make any changes to the JavaScript and CSS files as all paths are already set correctly.

5. In `themes/bar/templates/Layout/Page.ss`, just below `$Form` add:

```
<% if GalleryImages %>
  <section id="gallery">
    <% control GalleryImages %>
      <a class="cboxElement" title="$Title" rel="gallery"
       href="$ResizedBaseImage">
        $BaseImage.CroppedImage(125, 125)
      </a>
    <% end_control %>
  </section>
<% end_if %>
```

6. You can add some site-specific styling in `themes/bar/css/layout.css`. The gallery's general layout resides in the library's own CSS file, but don't change anything there as you might want to update it occasionally to the latest version.

7. Now you can reload the gallery page in the frontend, the general page should look like the screenshot at the start of this section. If you open a specific image, you'll see this modal window:



In the previous screenshot you can see a slight modification of the original code. To declutter the interface "`image {current} of {total}`" has been replaced with "`{current} / {total}`" in the copied JavaScript file. So now the text below the image is **32 / 33** instead of **image 32 of 33**. But this is just a question of personal preference.

If it doesn't work out as expected, check the following two things:

- In the source code of your page, inside the `<header>` section, the `colorbox.css` file should be referenced.
- Near the bottom of the page, your `jquery.colorbox-min.js` should be referenced.

If either of them is missing, check that you've referenced the correct path and that the file is where it should be. SilverStripe is intelligent enough not to include missing files in the final HTML of your page.

What just happened?

Our image gallery is very fancy, but how does it actually work?

We're creating a single set of images on our page, referenced by `gallery`. It's used both when setting the JavaScript plugin up and when connecting it with the relevant images in the template—coming up next.

Image gallery template

Instead of creating a dedicated `themes/bar/templates/Layout/GalleryPage.ss` we extended the layout's default `Page.ss`. If that file is already very long or you need to add a lot of specialized code, put your code in files of their own.

In our example everything is still pretty short and simple, so we'll add the gallery part to the existing page. And we don't need to copy any code or split it into various includes. But it always depends on the situation; you'll need to make a sensible decision in your projects.

What actually happens in this example's template is:

- Firstly, we check if there are gallery images. This excludes any non-image gallery pages and also galleries lacking any images.
- Then we iterate over all images with the `GalleryImages` control.
- We can then add all of the images as if we were only linking to the files themselves. ColorBox, using `rel="gallery"`, is taking care of the rest.
- For that we need two images—one thumbnail for a preview and one bigger image for full display.
- First we reference the bigger image. By using the `ResizedBaseImage()` method we're fetching the URL of the resized image.
- For the thumbnail we take a different approach—you don't always require a dedicated control structure. If you only need to scale an image and provide a full `image` tag (instead of simply the source URL), this is all you need:

  ```
  $BaseImage.CroppedImage(125, 125)
  ```

  You'll get a squared and cropped image.

`$BaseImage.CroppedImage(125, 125).URL` is not supported by the template engine. You can only get a single subelement through the dot operator. If you need to get the URL of a resized image (or its orientation, height, or width), you need a control structure. If you don't need to resize, `$BaseImage.URL` will work just fine.

- The example HTML output of a single `GalleryImage` looks like this:

  ```
  <a href="assets/Uploads/gallery/_resampled/SetWidth720-bar_
  gallery_01.jpg" rel="gallery" title="" class="cboxElement">
    <img alt="bar_gallery_01.jpg" src="assets/Uploads/gallery/_
  resampled/croppedimage125125-bar_gallery_01.jpg">
  </a>
  ```

Instead of `href="$ResizedBaseImage"` and the custom method behind it, we could have achieved the same just within the template:

```
href="<% control BaseImage %><% if Orientation == 2 %>$SetWidth(720).
URL<% else %>$SetHeight(719).URL<% end_if %><% end_control %>"
```

However, this is difficult to read and you can't add any whitespaces as the HTML wouldn't validate any more (spaces in `href="..."` are forbidden). Therefore, we've created our custom method in the `GalleryImage` class. Following the MVC pattern, the template calls this method; it's aware of the current context and returns the URL for the correct object.

As a rule of thumb, if the template is getting increasingly hard to read, think about replacing that part with some code in the Model or Controller. That's exactly what we've done in our example, after recognising that this is simply too much for the View on its own.

Although you might need to put in a little more effort into our gallery than with other systems, it really starts to shine when you need customizations. There are virtually no limits once you've set this up.

Transforming our code into a module

Creating an image gallery is pretty nice. We like it so much that we might reuse it in another project or even release it as an open source module at `http://silverstripe.org/modules`. However, copying the right parts from the `mysite/` folder is quite a headache—things might break or we might copy unneeded code. So let's simply create a module of our gallery right away, that's easy to reuse later on.

As the image gallery and intro page are closely related, we'll package both into our module.

Creating a module

As we've already said in the previous chapter, a widget is a specific module. So you know how to create a module:

- Create a new folder in the base directory.

- In this folder create a file `_config.php`, containing any configuration settings. If you don't have any, it's enough to put the opening PHP tag into it. However, you cannot leave the file out. SilverStripe will only inspect and include a folder if it contains a file named `_config.php`.
- Organize the rest of your code in subfolders like we already did—`code/`, `templates/`, and so on.

> Be sure to **move** any code from the previous example and not **copy** it. Otherwise SilverStripe will find duplicated classes, and as it doesn't know which is the "right" one, it will display an error message stating you have duplicate classes.

Let's create a new module folder `module_gallery/` including an "empty" configuration file as well as the subfolders `code/`, `css/`, `thirdparty/`, `templates/Includes/`, and `templates/Layout/`.

> **Naming conventions for modules**
>
> Using the prefix `module_` is not required, but as we're already using `widget_` it might be a good idea for keeping things organized and self-explaining. Although this isn't done by most modules, it's a good idea for marking our own modules.

This is the base structure of a module. Simply reuse it whenever you want to kick off a new module.

Required changes

Before we start: a module should be **self-contained**. That means you should only need to download it, place it inside the root folder, rebuild the database, and that's it. No code changes should be required for getting it up and running. We'll keep this in mind while at the same time making it easily extendable.

Time for action – adapting the module

We have all of the required code already. Now it's just a question of putting it in the right place and making minor changes to it. There are many steps to follow, but most of them are very simple, so let's get started.

1. Move over the following code files to your new module: `CustomImage.php`, `GalleryImage.php`, `GalleryPage.php`, and `IntroPage.php`. Don't copy, it will lead to errors.

2. Relocate the `mysite/thirdparty/colorbox/jquery.colorbox-min.js` and `mysite/thirdparty/colorbox/colorbox.css` files to our new module and keep them in the `thirdparty/` directory. Copying these won't lead to errors, but you're just keeping unneeded duplicates so it's best to move instead.

3. Move the images from `mysite/thirdparty/colorbox/images/` to `module_gallery/thirdparty/colorbox/images/`.

4. Create an include file `module_gallery/templates/Includes/Intro.ss` and move the `<figure>` part from `themes/bar/templates/Layout/IntroPage.ss` there. The new intro file should then contain:

```
<figure id="rotator">
  <ul>
    <% if CustomImages %>
      <% control CustomImages %>
        <li<% if First %> class="show"<% end_if %>>
          <a href="$Top.PageRedirect.URLSegment">
            <% control BaseImage %>
              <% control SetSize(720, 478) %>
                <img src="$URL" alt="Intro image number $Pos"
                  class="rounded transparent-nonie shadow"/>
              <% end_control %>
            <% end_control %>
          </a>
        </li>
      <% end_control %>
    <% end_if %>
  </ul>
</figure>
```

5. Create a file `module_gallery/templates/Layout/IntroPage.ss` and only add the include `Intro.ss`. It should simply contain:

```
<% include Intro %>
```

6. Create a generic `module_gallery/templates/Layout/GalleryPage.ss` page:

```
<section class="typography">
  $Content
  $Form
  <% include Gallery %>
</section>
```

7. Create the include we've just referenced (`module_gallery/templates/Includes/Gallery.ss`) and move the `<% if GalleryImages %>` part from `themes/bar/templates/Layout/GalleryPage.ss` there. The file should then contain:

```
<% if GalleryImages %>
  <section id="gallery">
    <% control GalleryImages %>
      <a class="cboxElement" title="$Title" rel="gallery" href="$R
esizedBaseImage">
        $BaseImage.CroppedImage(125, 125)
      </a>
    <% end_control %>
  </section>
<% end_if %>
```

8. Move the `SortableDataObject` line from `mysite/_config.php` to the module's configuration.

9. Set up a `MODULE_GALLERY_DIR` constant in the module's configuration file:

```
define('MODULE_GALLERY_DIR', 'module_gallery');
```

10. In `GalleryPage.php` and `IntroPage.php`, replace `PROJECT_DIR` with our new constant.

11. Move the required icons over.

12. In `GalleryPage.php`, add jQuery and correct the path to the CSS and JavaScript file:

```
Requirements::javascript(
  THIRDPARTY_DIR . '/jquery/jquery-packed.js'
);
Requirements::css(
  MODULE_GALLERY_DIR . '/thirdparty/colorbox/colorbox.css',
  'screen,projection'
);
Requirements::javascript(
  MODULE_GALLERY_DIR . '/thirdparty/colorbox/jquery.colorbox-min.
js'
);
```

13. In the `IntroPage.php` include jQuery and the rotator-specific styling and script:

```
Requirements::javascript(
  THIRDPARTY_DIR . '/jquery/jquery-packed.js'
);
Requirements::javascript(
  MODULE_GALLERY_DIR . '/javascript/rotator.js'
);
Requirements::css(
  MODULE_GALLERY_DIR . '/css/rotator.css', 'screen,projection'
);
```

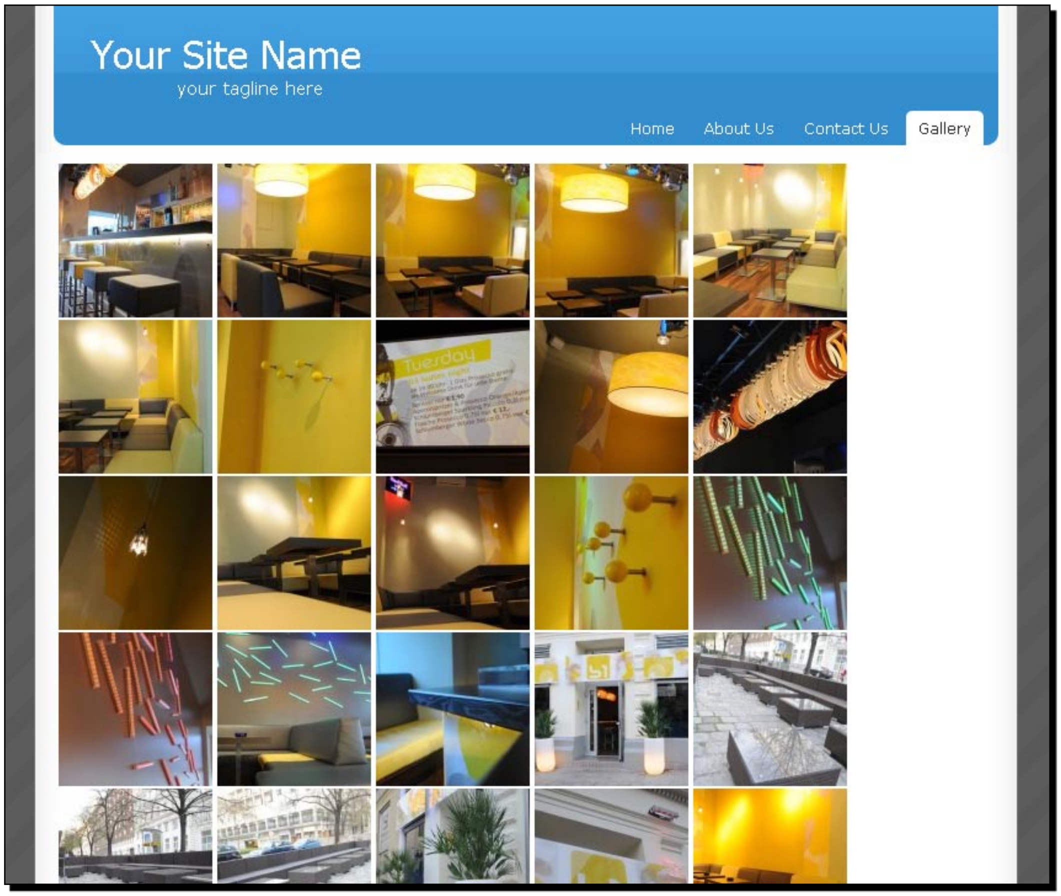
14. Create the CSS file we've just referenced, excluding `#rotator` itself, as that's needed in our theme and isn't generic:

```
#rotator ul li {
  float: left;
  position: absolute;
  list-style: none;
}
#rotator ul li img {
  padding: 10px;
}
#rotator ul li.show {
  z-index: 500;
}
#rotator a {
  text-decoration: none;
  border: 0;
}
```

15. Finally, move the rotator JavaScript from the Controller to the file we've just referenced. Separating the Controller and client-side code is something you should always try to achieve.

16. Congratulations! This is all you need to do to create our custom module.

If you create a new project, add the core files, the DOM and Uploadify modules, and our own, then the gallery page would look like this—using BlackCandy, SilverStripe's default theme:



What just happened?

We didn't change too much in the module itself. We moved the `SortableDataObject` to the configuration file and we set the path constant, CSS, and JavaScript straight. We've also added jQuery to both pages as we can't rely on the project-specific `Page` class to do it. SilverStripe doesn't include the same file more than once, so there is no potential for unnecessary statements.

Time for action - integrating the module into our page

Now that we've modularized our code, we need to tweak our templates to point to the module.

1. In your theme's folder, change the site-specific `IntroPage.ss` to:

```
<div><a href="$PageRedirect.URLSegment">
  <img src="$ThemeDir/images/background.jpg" alt="Background"
id="background"/>
</a></div>

<% include Intro %>

<aside id="bottom" class="transparent rounded shadow">
  <% include BasicInfo %>
</aside>
```

2. Copy `Page.ss` to `GalleryPage.ss` and replace the gallery code with an include, so the content part looks like this:

```
<section class="typography">
  $Content
  $Form

  <% include Gallery %>

  <aside id="sidebar">$SideBar</aside>
</section>
```

3. In `Page.ss`, remove the gallery code altogether so that the content part is as follows:

```
<section class="typography">
  $Content
  $Form
  <aside id="sidebar">$SideBar</aside>
</section>
```

4. Rebuild the database and flush.
5. Your page should now look just like before. We've simply added some clever modularization.

What just happened?

The most important part here is that we've put the module-specific template parts into includes. This allows us to perform some trickery in the site-specific templates. Remember the order in which template files are applied, for example, for the `GalleryPage` class:

1. `mysite/templates/Layout/GalleryPage.ss`
2. `themes/bar/templates/Layout/GalleryPage.ss` (assuming your theme is set to `bar`)
3. `module_gallery/templates/Layout/GalleryPage.ss`
4. `mysite/templates/Layout/Page.ss`
5. `themes/bar/templates/Layout/Page.ss`
6. And it's always embedded in `mysite/templates/Page.ss` or `themes/bar/templates/Page.ss` respectively

We're using this principle in two ways:

- First we're overwriting the module's layout pages with our theme's project-specific ones.
- But as we don't like duplicating the generic part, we've put that into includes. These includes don't exist in our theme, so we're falling back to the ones in the module. Nifty!

While we cannot use the layout's `Page.ss` for galleries any more as it's overwritten by the module's `GalleryPage.ss`, that's something you can't avoid when using a general and self-contained module.

Now our gallery and intro page are fully modularized. Great!

Contribute back to the community

You'll most likely have already seen that there are many interesting modules available for download at `http://silverstripe.org/modules`.

Now that we've created our own module, it would be only fair to let others enjoy it as well—just like we're benefiting from the core product and various extensions. And you are likely to get bug reports, patches, documentation, or even feature extensions back in return.

In order to be useful to others, keep the following points in mind:

- See if there is already a very similar module. If there is, try joining forces. It should lead to better results and be more useful for everyone.
- Make sure your module is as generic as possible. It shouldn't just be useful for your specific use case.

- Stick to SilverStripe's conventions, see `http://doc.silverstripe.org/coding-conventions`
- Document your code.
- Maintain your project; it should always be working with the latest stable version of SilverStripe.

Those are the most important points; for a complete list take a look at `http://doc.silverstripe.org/module-maintainers`.

[At the moment we are ignoring the requirement to keep our code localizable and translatable, but we'll come back to that a little later.]

Summary

We've only been able to take a look at some of the most important features of the DataObjectManager and Uploadify modules. There's a lot more you can do with them. Uploadify, for example, can be used to store files on Amazon's Simple Storage Service (S3), which is a distributed, scalable storage system capable of dealing with large amounts of files and traffic.

For full details on these two modules, take a look at their documentation:

- `http://www.leftandmain.com/silverstripe-modules/2010/08/23/dataobjectmanager/`
- `http://www.leftandmain.com/silverstripe-modules/2010/08/26/uploadify/`

Additionally we've learned a lot about:

- How to integrate DataObjectManager and Uploadify into our page
- Handling images in SilverStripe
- How to create a fully customizable intro and gallery page

And we had a look at how to create our own modules.

That's already quite a lot, but it lacks any user interaction. We'll focus on that in the next two chapters, taking our page from a one-way presentation to an interactive communication channel.

8
Introducing Forms

This chapter, we'll focus on user interaction. How do you do that? By using forms.

There are two main use cases we want to cover: providing a contact form and letting the users book a table at our bar. Both features will rely on sending out the input as e-mails.

For achieving that, we'll cover:

- Creating and displaying a form
- Getting and processing the user's response
- Forward the content via e-mail
- Provide client-side validation for quicker feedback

There are many new things waiting for being discovered, so let's get started.

There is a very good module for creating forms via the CMS interface, putting everything together graphically. You can find it at `http://www.silverstripe.org/user-forms-module`. While the user forms module is very powerful, it has its limitations and that's why we're covering how to manage our forms in code ourselves. It's more work, but there are hardly any limits. For your use cases, you should evaluate which approach fits your needs best.

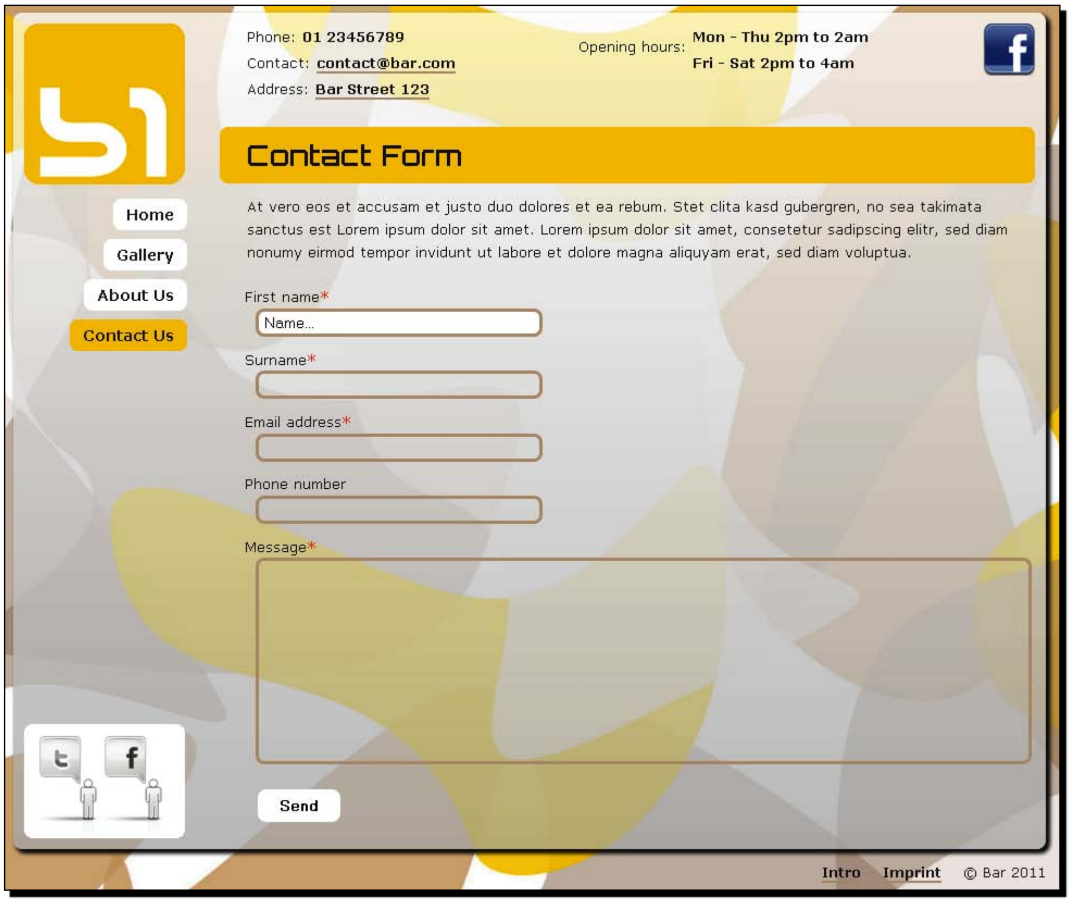
Building a contact form

Isn't there a finished contact form we simply need to activate? No—following SilverStripe's general approach, there isn't.

You're provided with the tools to easily build one yourself, but there isn't a default one as the requirements vary too much. One cannot build a simple contact form that satisfies all needs.

Time for action – creating a basic contact form

Our contact page is in desperate need of a contact form. As there isn't one available on our site, let's build it ourselves. It should be added on a new page and should allow users to provide their name, e-mail address, a phone number, and a message. After submitting the form the user should see a thank-you message while the e-mail is actually sent to the manager. E-mails will be sent as HTML to allow nice formatting.



Phone: 01 23456789
Contact: contact@bar.com
Address: Bar Street 123

Opening hours: Mon - Thu 2pm to 2am
Fri - Sat 2pm to 4am

Contact Form

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

First name*: Name...
Surname*
Email address*
Phone number
Message*

The backend

Let's start off by creating the backend. We need to add two new fields: one to define which e-mail address the message should be sent to; the other one to provide a success message (customizable via the CMS) after submitting the form.

Additionally, we need to set up the form, process it once it's submitted, and finally send the e-mail.

1. Create a new Model in `mysite/code/ContactPage.php`:

   ```
   class ContactPage extends Page {

     public static $db = array(
       'Mailto' => 'Varchar(100)',
       'SubmitText' => 'HTMLText',
     );

     public function getCMSFields(){
       $fields = parent::getCMSFields();
       $fields->addFieldToTab(
         "Root.Content.Email",
         new EmailField('Mailto', 'Recipient')
       );
       $fields->addFieldToTab(
         "Root.Content.Email",
         new HTMLEditorField(
           'SubmitText',
           Text after sending the message'
         )
       );
       return $fields;
     }

   }
   ```

2. Create the Controller:

   ```
   class ContactPage_Controller extends Page_Controller {

     public static $allowed_actions = array(
       'sent',
     );

     protected function Form(){
       define('SPAN', '<span class="required">*</span>');
       $firstName = new TextField('FirstName', 'First name ' . SPAN);
       $firstName->addExtraClass('rounded');
       $surname = new TextField('Surname', 'Surname' . SPAN);
       $surname->addExtraClass('rounded');
   ```

```
    $email = new EmailField('Email', 'Email address' . SPAN);
    $email->addExtraClass('rounded');
    $phone = new TextField('Phone', 'Phone number'
    $phone->addExtraClass('rounded');
    $comment = new TextareaField('Comment','Message' . SPAN);
    $comment->addExtraClass('rounded');
    $fields = new FieldSet(
      $firstName,
      $surname,
      $email,
      $phone,
      $comment
    );
    $send = new FormAction('sendemail', 'Send');
    $send->addExtraClass('rounded');
    $actions = new FieldSet(
      $send
    );
    $validator = new RequiredFields(
      'email',
      'comment',
      'firstName',
      'surname'
    );
    return new Form($this, 'Form', $fields, $actions, $validator);
  }

  public function sendemail($data, $form){
    if(!empty($this->Mailto)){
      $email = $this->Mailto;
    } else {
      $email = EMAIL;
    }
    $from = $data['Email'];
    $to = $email;
    $subject = "Contact Form Bar";
    $email = new Email($from, $to, $subject);
    $email->setTemplate('Email');
    $email->populateTemplate($data);
    $email->send();
    Director::redirect($this->Link('sent'));
  }

  public function sent(){
    return array();
  }
```

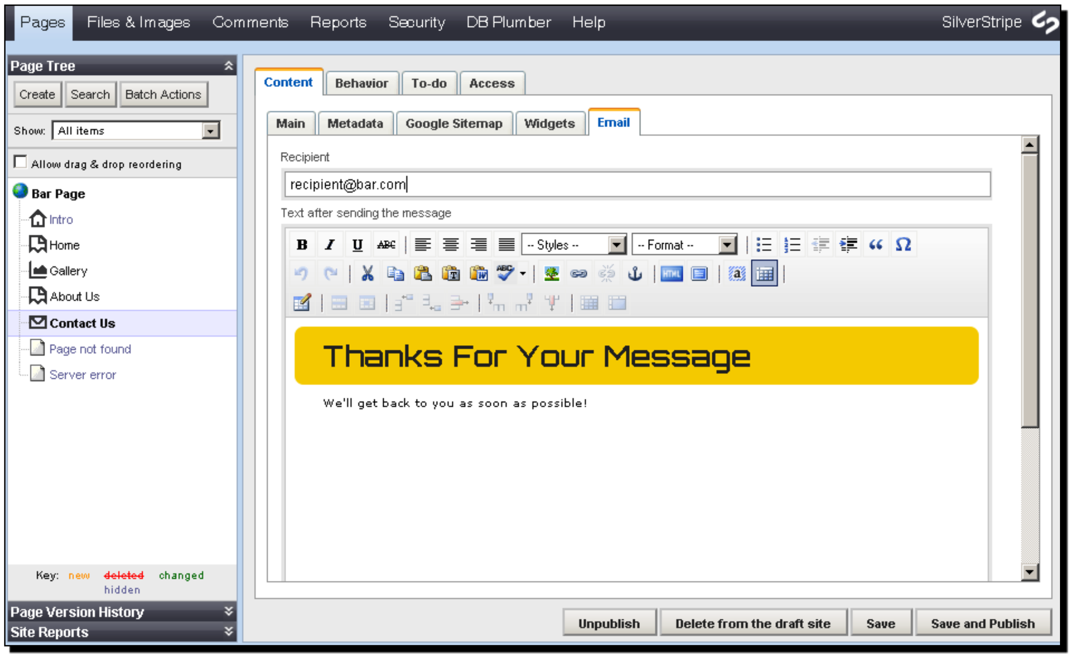
```
    protected function IsSuccess(){
      $url = Director::URLParams();
      return (isset($url['Action']) && ($url['Action'] == 'sent'));
    }

  }
```

3. We set up the `EMAIL` constant some time ago in `mysite/_config.php` and are reusing it here. Check that you have defined it—the snippet we discussed looked like this:

```
if(Director::isLive()){
  define('EMAIL', 'office@bar.com');
} else {
  define('EMAIL', 'admin@bar.com');
  Email::send_all_emails_to(EMAIL);
}
Email::setAdminEmail(EMAIL);
```

4. Rebuild the database.

5. Create a **Contact Page** in the CMS. It should look like this:



Recipient

recipient@bar.com

Text after sending the message

Thanks For Your Message

We'll get back to you as soon as possible!

[As in the previous chapters we've added some styling so that the pages are a little more appealing to the visitor.]

Including our form in the frontend

Now that the backend has been completed, we still need to include our form in the frontend, so that we can actually use it. But that's only four new lines of code and we're done.

1. In `themes/bar/templates/Layout/Page.ss` replace the typography section with the following code. Only the highlighted part is new, the rest should already be present from the previous chapters:

```
<section class="typography">
  <% if IsSuccess && SubmitText %>
    $SubmitText
  <% else %>
    $Content
    $Form
  <% end_if %>
  <aside id="sidebar">$SideBar</aside>
</section>
```

2. Reload the page in the frontend.
3. It will look rather dull, but by adding some style to `themes/bar/css/form.css` it might look something like the image given at the start of this section.

Creating an e-mail template

The website itself is finished. But we still need the e-mail template to actually send the information to our intended recipient.

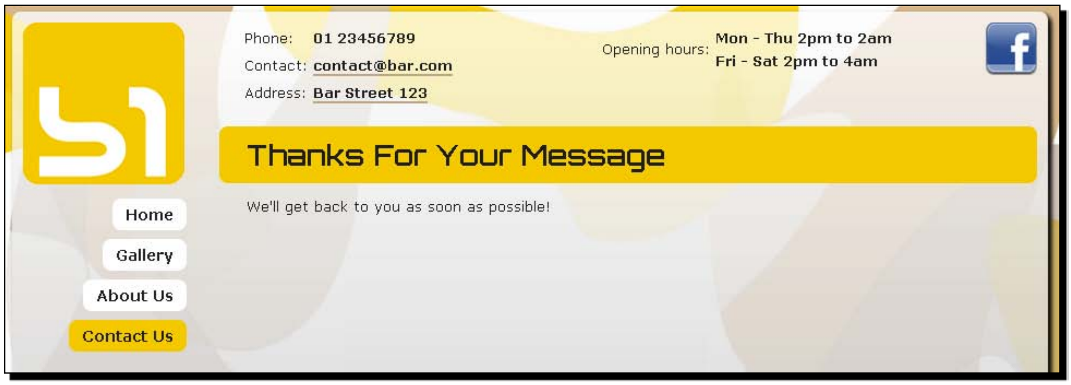
1. Create a new template file in `themes/bar/templates/ContactEmail.ss` (directly there, not in `Includes/` or `Layout/`):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <style>
      p, li {
        font-size: 1em;
        color: #000;
      }
    </style>
  </head>
  <body>
    <ul>
      <% if FirstName %><li>First name: $FirstName</li><% end_if
%>
      <% if Surname %><li>Surname: $Surname</li><% end_if %>
      <% if Email %><li>Email address: $Email</li><% end_if %>
      <% if Phone %><li>Phone number: $Phone</li><% end_if %>
      <% if Comment %><li>Message: $Comment</li><% end_if %>
    </ul>
    <p>You can simply reply to this message.</p>
  </body>
</html>
```

2. Send a test message by filling in the form on the newly created page, which has the URL `/contact-us`.

> For this to succeed, you must have PHP's mail function set up correctly. While this is the case for pretty much all hosting providers, you need to take care of it on your own development machine. If you're using a complete web server package such as XAMPP (`http://www.apachefriends.org/en/xampp.html`), an SMTP server should already be included. But in manual setups you'll need to install you own. You can find an extensive list at `https://secure.wikimedia.org/wikipedia/en/wiki/List_of_mail_servers`. We'll assume that everything is configured as required.

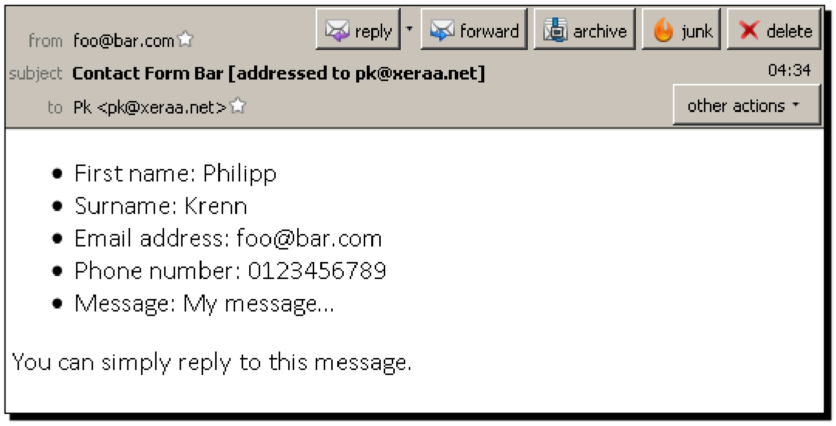
3. After sending the message, you should be redirected to a page stating the success of your action—the default header, footer, and menu are the same as on any other page:


Phone: 01 23456789
Contact: contact@bar.com
Address: Bar Street 123
Opening hours: Mon - Thu 2pm to 2am
Fri - Sat 2pm to 4am

Thanks For Your Message

We'll get back to you as soon as possible!

4. The message should then look something like this in your e-mail program of choice:



from foo@bar.com
subject **Contact Form Bar [addressed to pk@xeraa.net]**
to Pk <pk@xeraa.net>
04:34

- First name: Philipp
- Surname: Krenn
- Email address: foo@bar.com
- Phone number: 0123456789
- Message: My message...

You can simply reply to this message.

5. As the note at the bottom implies, you can simply click on the **reply** button to answer to the original sender (assuming that he or she has provided his or her correct e-mail address).

What just happened?

Quite a lot. Let's start off with the easy parts and advance slowly.

The frontend

`$Form` has so far only been replaced by the login form, when accessing `/admin`. Now it's also replaced by our custom contact form. There's nothing else for us to do here, all the magic happens in the Controller, which we'll come to in a moment.

For providing instant feedback to our visitors, we've introduced:

```
<% if IsSuccess && SubmitText %>
  $SubmitText
<% else %>
```

We've defined a new text `$SubmitText` in the CMS for display after successfully sending a message. In case such a text exists and a message has just been sent, we're displaying it. This provides a far better user experience than simply swallowing the text, as the user will know it's been processed.

Why is there no ContactPage.ss?

Note that we're using a fallback again. There is no `ContactPage.ss` in the layout folder, so we're using `Page.ss`. It helps keeping the number of files and lines of code down, so it's easier to maintain.

The e-mail template

E-mail clients support HTML messages, but they do so rather poorly. So instead of providing a full blown version of our page, we're using a simple but functional template. And we're not relying on the latest and greatest HTML5 version, but rather XHTML. The HTML support of the various e-mail clients is often pretty poor, so we'll try to avoid any problems and keep it simple and traditional.

Whether you should really use HTML in e-mails , or just plain text, is the subject of fierce debates. In our case it enhances the visual presentation and comes from a trusted source, our own page, we have full control over the HTML and can escape any user input. Therefore we deem it admissible.

Rather than embedding the e-mail in the default `themes/bar/templates/Page.ss`, we're creating a custom version in the same folder: `ContactEmail.ss`. We then need to explicitly call this file in the Controller, so it's being used.

In the template, we simply check if each value has been set and then add it to our e-mail if it has. The only thing to watch out for is that the fields in the Controller (when setting up the form) and the template must have the same name. As we're having `$FirstName` in our template, we must also call the field like that in the Controller.

The Model

In the Model, we add a field for the mail's recipient. That way it's configurable in the CMS—nice!

Additionally, we provide a field for the message displayed after successfully sending an e-mail. We've already added it in the template. Simply blanking the inputs is not a good solution; the user can never be sure if their message has really been sent or if an error occurred.

That's it, there is nothing else new here.

The Controller

The Controller is split up into three different functions, plus `$allowed_actions`. We've already met this static variable: it limits the actions (specifically the methods in the Controller) a user can call. Note that form actions (like our function `sendemail()`) shouldn't be included in the `$allowed_actions` array. You might still see this in older examples as it was only changed recently, but it's now discouraged.

Setting up the form

We create the form itself with the function `Form()`. You can name it differently, but if you only have a single one on a page, this is a pretty obvious choice. It's not a convention, but still advisable.

Our function returns all of the HTML necessary for rendering a form. Specifically, it's a form object used by SilverStripe for this exact purpose.

Do you remember the form fields from the Model chapter, for example, `TextField`, `TextareaField`, and so on? So far we've only used them in the backend, but they are also used for generating the frontend. The usage is also the same: first define the field we want to use and then add an optional label, making this as easy as possible.

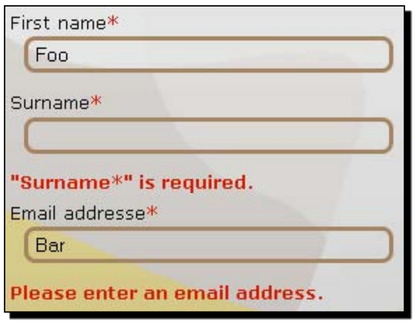
So we're setting up a field containing the first name and labeling it. Then we're adding a CSS class that is only used for styling purposes:

```
$firstName = new TextField(
  'FirstName',
  'First name<span class="required">*</span>'
);
$firstName->addExtraClass('rounded');
```

Although the CSS isn't essential for the core functionality of our example, it shows how easy it is to extend forms in SilverStripe. For a complete list of available methods take a look at `http://api.silverstripe.org/2.4/forms/core/FormField.html`.

> Before you ask why we didn't simply apply the CSS to all fields: we only want to define the styling once, so if we want to change anything there's just a single place to edit. As CSS doesn't support Object-Oriented style inheritance (without relying on some additional tools), we set up a dedicated styling class and apply it to all elements. You can, of course, take a different approach.

We'll set up the other fields in just the same way, namely surname, e-mail, phone, and comment. Note that there is a special field for e-mail addresses, making sure people can only enter a valid e-mail: `EmailField()`. SilverStripe will automagically take care of this. In case you don't play by the rules, you'll get an error message. You can also see what happens when you leave a required field empty, for example, surname:



First name*

Foo

Surname*

"Surname*" is required.

Email addresse*

Bar

Please enter an email address.

Then we're putting all our fields into a `FieldSet()`. That is simply a container for the fields. Could we've also added the fields directly there? Absolutely, but then we couldn't have added the custom CSS class. In case you don't need it, you could have simply written:

```
$fields = new FieldSet(
  new TextField('FirstName', 'First name' . SPAN),
  new TextField('Surname', 'Surname' . SPAN),
```

Be careful with commas

`FieldSet()` is an object, so you must put a comma between individual fields but not after the last one, otherwise PHP will throw a syntax error.

This is very different to arrays where you're free to add an optional comma at the end. We're generally adding the trailing comma as a convention, so if you add another element, you don't need to pay attention to the previous line.

After adding the fields, we also need an action for sending the whole form—generally a button you can click. It's called `FormAction()`, but otherwise it's used just like fields. We could possibly have more than button, but here we're satisfied with a single one. We still need to wrap it into a `FieldSet()`.

Next we're defining which fields are required before the form can be submitted. So inside the `RequiredFields()` we're passing each field name (the first argument when creating them) that we've labeled for being required. So only the phone number can be left empty, all other fields can't. SilverStripe will take care of that.

Finally, having set up the individual parts, we can put all of this into action. We're returning a `new Form()`. This method is always called `Form()`, independent from the calling method's name. The available parameters are:

- The name of the controller responsible for processing this form. As this is defined in the same class, we can simply use `$this` and it is set up correctly.
- The second argument is the name of the method in which the `Form()` is created—in our case also called `'Form'`. You must give them the same name, otherwise your form won't work!
- The fields we've created. You can, of course, also just create the `FieldSet()` and fields here—whatever you find more intuitive.
- The actions of this form.
- The optional validator.

In our case it's:

```
return new Form($this, 'Form', $fields, $actions, $validator);
```

In case you have called your method `ContactForm()` instead of `Form()`, it will be:

```
return new Form($this, 'ContactForm', $fields, $actions, $validator);
```

So don't get confused with the different `Form` elements.

Processing the input

Once the user has filled out and submitted the form, it will be processed by `sendemail($data, $form)`. That's the function we've defined for this task in `new FormAction('sendemail', 'Send')`. In order to avoid going too much back and forth, here is the content of the method again:

```
if(!empty($this->Mailto)){
  $email = $this->Mailto;
} else {
  $email = EMAIL;
}
$from = $data['Email'];
$to = $email;
$subject = "Contact Form Bar";
$email = new Email($from, $to, $subject);
$email->setTemplate('Email');
$email->populateTemplate($data);
$email->send();
Director::redirect($this->Link('sent'));
```

`$data` contains the values of the filled out form and `$form` the data object itself. So you can simply access the sender's e-mail field through `$data['Email']`.

For sending the data to the correct recipient, we're first checking if a form-specific address has been set up. If it hasn't, we fall back to the one defined in `mysite/_config.php`.

Next we send the e-mail and if everything has worked out, we're redirecting to a subpage of the current page. So if you were on `/contact` you would then be redirected to `/contact/sent`. On that page we'll inform the user that the message has been sent successfully.

With `$this->Link()` you'll get the current URL. If you provide the optional argument, the given string is added as the action to which you want to go on the current page. In other words the method appends `sent` to the current page.

Sending the e-mail

First, we're creating a new e-mail object, containing sender, recipient, and subject: `new Email($from, $to, $subject);`

Then, using `setTemplate()`, we're instructing the e-mail object not to use `themes/bar/templates/Page.ss` but `themes/bar/templates/ContactEmail.ss` instead.

Before actually sending the e-mail, we need to add the current information. You can do that with `populateTemplate($data)` so that `$data['Email']` is accessible via `$Email` in our `ContactEmail.ss`.

Handling the success page

For handling the success message, we need two functions: `sent()` for setting up the URL and `IsSuccess()` for handling it in the template.

In order to make `/contact/sent` available, we need a public `sent()` method on the `/contact` page. For simplicity, we're again falling back to the default `Page.ss` so there's nothing else we need to do—simply return an empty array.

In `Page.ss`, we'll obviously need to check if we're on such a `/sent` subpage. And if a `$SubmitText` has been set, we'll display it.

For accessing the URL use: `Director::URLParams()`. It's a named array with the following elements (in order; the first two are required):

Array element	Description
`Controller`	The Controller mapping the initial request—not necessarily the Controller rendering the final page.
`URLSegment`	The first "part" of the URL, for example, `about-us`.
`Action`	The second "part", which often maps to a method in the Controller, for example `sent`. This element and the following ones are optional and don't need to exist.
	If you're using nested URLs, such as `/level1/level2`, this would be `level2`.
`ID`	The third "part", often an ID of the specific page, for example `/entries/fetch/20`, which would probably fetch the element 20 on the page `entries`.
`OtherID`	The fourth "part", any other element in case your page is nested that deeply.

For our page `/contact/sent` the actual array is:

```
array(5) {
  ["Controller"]=>
  string(17) "ModelAsController"
  ["URLSegment"]=>
  string(7) "contact"
  ["Action"]=>
  string(4) "sent"
  ["ID"]=>
  NULL
  ["OtherID"]=>
  NULL
}
```

So it fetches the current URL segment and, if set, the following three parameters. In our case we're just interested in the first one, the action. Using that we tell the template if we're currently on a `/sent` subpage or not.

Convention of method names

In case you haven't noticed the naming conventions of the methods:

- `Form()` is UpperCamelCased as it's used in the template
- `sendemail()` as well as `sent()` are all lowercased as they are Controller actions
- `IsSuccess()` is a method for the Controller, so it's the same as the first bullet point

And that's all there is to do, not so hard after all—right?

Adding spam protection

You might want to add some spam protection to your contact form. Take a look at SilverStripe's spam protection module (`http://www.silverstripe.org/spam-protection-module`), which provides the basis for popular solutions such as Mollom or Recaptcha.

Renting a table

Now that sending contact forms has worked out so well, let's continue with the booking of a table—which is just another e-mail form in our application. But isn't there some code we can reuse instead of starting all over again? Copying big chunks of code isn't such a great idea after all—remember DRY.

If you think about it, it's obvious that the Model would be the same. In the Controller `sendemail()`, `sent()`, and `IsSuccess()` are so generic that we can reuse them as they are. Only `Form()` is different—using different fields. But how should we implement that?

Going "abstract"

In case you're not familiar with the `abstract` concept of object oriented programming:

- If something is `abstract`, it's only a preview and must be specifically implemented by an extending element. So it's basically a specific form of subclassing.
- Both methods and classes can be `abstract`.
- `abstract` methods must be overwritten in the extending class.
- Classes containing `abstract` methods must be declared `abstract` themselves.
- An `abstract` class can contain both abstract and concrete methods.

That sounds awfully complicated, so is it really worth it? Actually it's not as complicated as it might sound, and furthermore it's really clever and helpful. Let's extend our previous example with this concept to easily add the booking of a table.

Time for action – extending our form with abstraction

The general plan is to create an `abstract` class, which we'll then `extend` in two concrete classes **contact** and **rent form**. At the same time we'll hardly need to change anything in the templates, so let's get on with it.

The backend

1. Create a new file `mysite/code/Emailer.php`. Simply move the Model from the previous example here:

```
define('EMAILER_ICON_PATH', PROJECT_DIR . '/icons/mail');

class Emailer extends Page {

  public static $db = array(
    'Mailto' => 'Varchar(100)',
    'SubmitText' => 'HTMLText',
  );

  public static $icon = EMAILER_ICON_PATH;

  public function getCMSFields(){
    $fields = parent::getCMSFields();
    $fields->addFieldToTab(
      "Root.Content.Email",
      new EmailField('Mailto', 'Recipient')
    );
    $fields->addFieldToTab(
      "Root.Content.Email",
      new HTMLEditorField(
        'SubmitText',
        'Text after sending the message'
      )
    );
    return $fields;
  }

}
```

Don't forget to add the icon. We've already covered this in Chapter 4 while exploring the Model.

2. And the Controller becomes:

```
abstract class Emailer_Controller extends Page_Controller {

  public static $allowed_actions = array(
    'sent',
  );

  protected $subject = "Email Form Bar";

public function init(){
    parent::init();
    define('SPAN', '<span class="required">*</span>');
  }

  abstract protected function Form();

  public function sendemail($data, $form){
    if(!empty($this->Mailto)){
      $email = $this->Mailto;
    } else {
      $email = EMAIL;
    }

    $from = $data['Email'];
    $to = $email;
    $subject = $this->subject;
    $email = new Email($from, $to, $subject);
    $email->setTemplate('ContactEmail');
    $email->populateTemplate($data);
    $email->send();

    Director::redirect($this->Link('sent'));
  }

  public function sent(){
    return array();
  }

  protected function IsSuccess(){
    $url = Director::URLParams();
    return (isset($url['Action']) && ($url['Action'] == 'sent'));
  }

}
```

3. The `ContactPage` class then becomes:

```
class ContactPage extends Emailer {

  public static $hide_ancestor = 'Emailer';

}

class ContactPage_Controller extends Emailer_Controller {

  protected $subject = "Contact Form Bar";

  protected function Form(){
    $firstName = new TextField('FirstName', 'First name' . SPAN);
    $firstName->addExtraClass('rounded');
    $surname = new TextField('Surname', 'Surname' . SPAN);
    $surname->addExtraClass('rounded');
    $email = new EmailField('Email', 'Email address' . SPAN);
    $email->addExtraClass('rounded');
    $phone = new TextField('Phone', 'Phone number');
    $phone->addExtraClass('rounded');
    $comment = new TextareaField('Comment','Message' . SPAN);
    $comment->addExtraClass('rounded');
    $fields = new FieldSet(
      $firstName,
      $surname,
      $email,
      $phone,
      $comment
    );
    $send = new FormAction('sendemail', 'Send');
    $send->addExtraClass('rounded');
    $actions = new FieldSet(
      $send
    );
    $validator = new RequiredFields(
      'email',
      'comment',
      'firstName',
      'surname'
    );
    return new Form($this, 'Form', $fields, $actions, $validator);
  }

}
```

We didn't call the class EmailBasePage or EmailPage because, as an abstract, you can't use it as a page directly. You're encouraged to follow this naming schema for clarity, but it doesn't have any implications if you don't.

4. The RentPage class in mysite/code/RentPage.php looks very similar:

```
class RentPage extends Emailer {
}

class RentPage_Controller extends Emailer_Controller {

  protected $subject = "Reservation Form Bar";

  protected function Form(){
    $datetime = new TextField(
      'Datetime',
      'Date and time <noscript>
                     <br/> Format: YYYY-MM-DD HH:MM</noscript>' .
SPAN);
    $datetime->addExtraClass('rounded');
    $firstName = new TextField('FirstName', 'First name' . SPAN);
    $firstName->addExtraClass('rounded');
    $surname = new TextField('Surname', 'Surname' . SPAN);
    $surname->addExtraClass('rounded');
    $email = new EmailField('Email', 'Email address' . SPAN);
    $email->addExtraClass('rounded');
    $phone = new TextField('Phone', 'Phone number' . SPAN);
    $phone->addExtraClass('rounded');
    $count = new TextField('Count', 'Number of people' . SPAN);
    $count->addExtraClass('rounded');
    $comment = new TextareaField('Comment','Optional Comment');
    $comment->addExtraClass('rounded');
    $fields = new FieldSet(
      $datetime,
      $firstName,
      $surname,
      $email,
      $phone,
      $count,
      $comment
    );
```

```
        $send = new FormAction('sendemail', 'Send');
        $send->addExtraClass('rounded');
        $actions = new FieldSet(
          $send
        );
        $validator = new RequiredFields(
          'email',
          'firstName',
          'surname',
          'count',
          'datetime',
          'phone'
        );
        return new Form($this, 'Form', $fields, $actions, $validator);
      }

    }
```

Changes in the template file

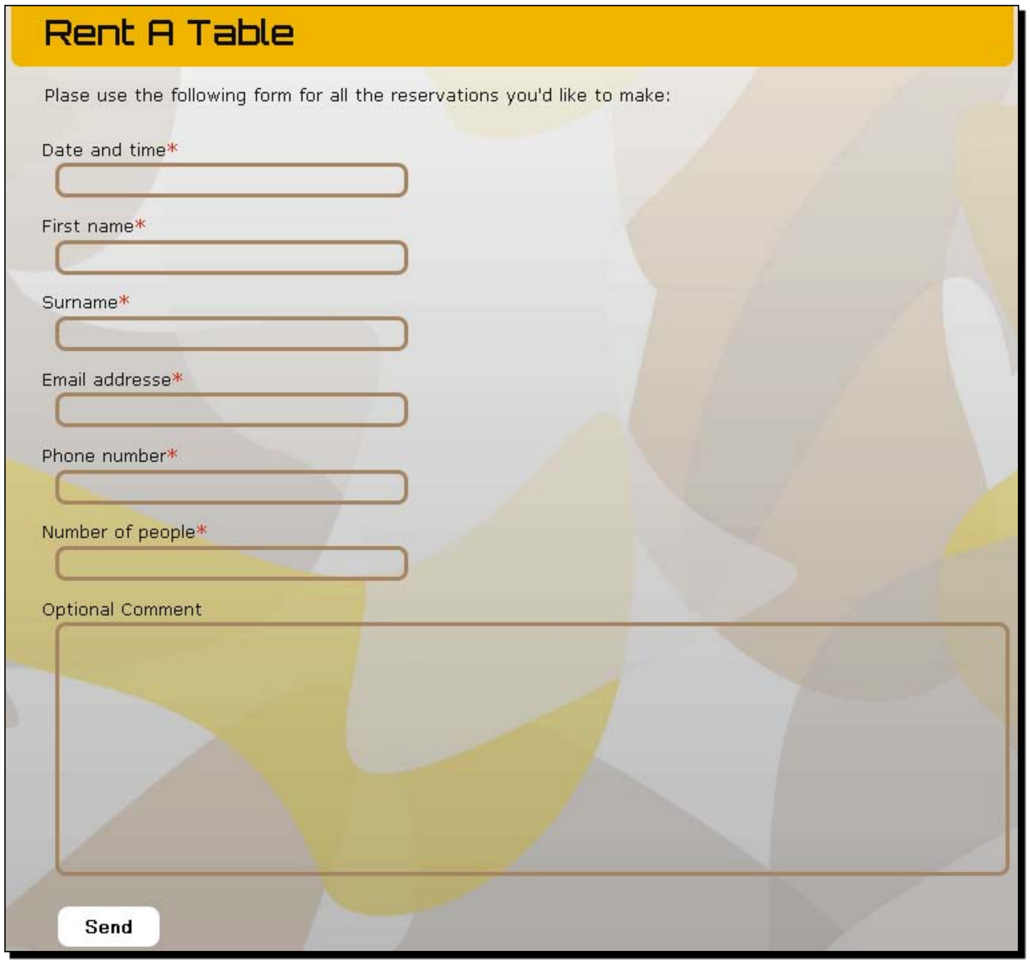
No changes are required in the template file, it can simply be reused as is.

Update the e-mail template

1. Rename the e-mail template from `ContactEmail.ss` to `Email.ss` for consistency.
2. Add the new fields `$Datetime` and `$Count` to the list:

```
<ul>
  <% if Datetime %><li>Date and time: $Datetime</li><% end_if %>
  <% if FirstName %><li>First name: $FirstName</li><% end_if %>
  <% if Surname %><li>Surname: $Surname</li><% end_if %>
  <% if Email %><li>Email address: $Email</li><% end_if %>
  <% if Phone %><li>Phone number: $Phone</li><% end_if %>
  <% if Count %><li>Number of people: $Count</li><% end_if %>
  <% if Comment %><li>Message: $Comment</li><% end_if %>
</ul>
```

While the contact page will just look the same as before, the new rent table page now is being displayed in a similar way:



Rent A Table

Plase use the following form for all the reservations you'd like to make:

Date and time*

First name*

Surname*

Email addresse*

Phone number*

Number of people*

Optional Comment

What just happened?

There were only some subtle changes to the existing code.

The templates are pretty clear, but why didn't we call the template `Emailer.ss` just like the abstract class where it's used? This would render the concrete form pages with this template—which we obviously don't want. The contact page inherits from `Emailer`, which in turn inherits from `Page`. Therefore, the templating engine will try to load `themes/bar/templates/Email.ss` instead of `Page.ss` in the same folder as the general layout file. The solution is simple: name the base template differently from the inherited class—`Email.ss` for example.

Abstract

First we'll dissect the `abstract` code, followed by the concrete implementation.

Model

We can reuse the Model as it is. We only need to rename the class as we've moved it to the general `Emailer`. We don't declare the Model `abstract` as there's nothing here that can be abstracted. If we need to add anything inside an implementation of our `Emailer`, we're free to do so by simply subclassing it. Don't worry; we'll cover that in another example.

Controller

We'll have an `abstract` method, so the class itself must be `abstract` as well.

This method is defined as: `abstract protected function Form();`

Although the other (concrete) methods can be reused in the subclasses, the `abstract` declaration must be implemented in them. We state that there must be a `protected function Form(){ ... }`, but not what it actually does.

For better usability, we need to have different subjects in the e-mails. But how can this be achieved when we use a method implemented in the base class? The solution is to declare an object variable `$subject` (or whatever you want to call it), which is then used via `$this->subject` in the method. In a subclass you overwrite the variable and it is then used in the base method. You can, of course, add it to the Model and make it accessible in the CMS, but we wanted to try out something new and different.

You can't declare a variable called `abstract`. It's syntactically not possible and also not necessary as it's automatically overwritten by a subclass.

Note that we have defined our SPAN constant in the abstract Controller as well. This way we only need to set it up once and can then use it in all of the implementing classes.

Implementation

The `ContactPage` now extends the `Emailer` class. As we don't need to add anything in the Model, we can leave it empty, except for `$hide_ancestor`. You wouldn't want CMS editors to accidentally add an abstracted page, which couldn't do anything useful, would you?

We define the specific subject we want to use in this class in the Controller. The `Form()` method is exactly the same as before, while we can leave out the other parts of the class—they are automatically inherited from the parent class.

RentPage is very similar, only with a few different fields. As you can see, we've duplicated the functionality without duplicating the code. Sweet!

Would this make another good module?

Most likely no. Although the contact form itself is pretty generic, renting a table is very specific. Reusing the code as it is, would happen only rarely. Therefore, there isn't too much of a motivation to pack it into its own module. Your mileage may vary, but we'll simply leave it where it is.

Adding client-side validation

SilverStripe already does the (most important) server-side validation, so why should we bother with adding client-side validation too? There are several reasons:

- A user gets the response a lot faster as everything is done within the browser. For server-side manipulation to kick in, the data must be sent to the server, validated there and the response sent back, before the user can be notified.
- You can reduce both network and server load as most errors are already caught in the user's browser—without needing to get back to your infrastructure.
- It's easier to help the user do it right. Most users don't do things wrong intentionally (though don't forget that there are always some bad guys), you should try to help them as much as possible.

Client-side validation today means using JavaScript. Don't be afraid, you won't need to use raw JavaScript as we'll rely on some clever tools.

Default client-side validation

At the moment they are disabled, but SilverStripe provides default client-side validators. They do the same as the server-side validation, so let's enable it:

- In `mysite/_config.php`, delete or comment out the following line:

  ```
  Validator::set_javascript_validation_handler('none');
  ```

- In `mysite/code/Page.php`, delete or comment out the following lines:

  ```
  Requirements::block(THIRDPARTY_DIR . '/prototype/prototype.js');
  Requirements::block(THIRDPARTY_DIR . '/behaviour/behaviour.js');
  Requirements::block(
    SAPPHIRE_DIR . '/javascript/prototype_improvements.js'
  );
  Requirements::block(
  ```

```
    SAPPHIRE_DIR . '/javascript/ConfirmedPasswordField.js'
  );
  Requirements::
    block(SAPPHIRE_DIR . '/javascript/ImageFormAction.js');
```

Now you can reload a form in the frontend. If you type "Foobar" into an e-mail field and click into another field, straight away you'll get the message: **Please enter an email address.** So the JavaScript validation kicks in as soon as you've finished editing a field and you can immediately correct any error. As soon as you click on the **Send** button and any required fields are still blank, you'll get error messages and the form is not submitted—it's waiting for your corrections. Now that's a major step forward in usability!

That's it, there is nothing else you need to do. SilverStripe automagically does the necessary steps according to the field type you've defined and whether a field is required or not.

> We intentionally disabled SilverStripe's default client-side validation in the previous chapters, so that we could first see the server-side validator. By default, both are enabled which is why you need to comment out the disabling code to get it back.

Enhanced client-side validation

SilverStripe's validation has the huge advantage of being automagically generated and added to your page. However, this has the drawback of a lack of control over the validators. In order to get that, first undo the steps we just did—so that no client-side validation is active at all. Otherwise, we'd have conflicts between the two approaches.

There are many plugins you can use, but we'll go with the excellent Validation plugin for jQuery, which you can find at `http://bassistance.de/jquery-plugins/jquery-plugin-validation/`. It's obviously based on jQuery like the rest of our website's frontend, is easy to use but still powerful and quite elegant.

Time for action – using jQuery's Validation plugin on the contact page

Let's get started with it, first using it on our contact page. We need to add the basic JavaScript library and then some custom code for our form fields to provide our customized validation.

1. Download the plugin and add it to the `Emailer_Controller` class. SilverStripe's default JavaScript validators should still be disabled to avoid any collisions:

```
public function init(){
  parent::init();
  Requirements::javascript(
    PROJECT_DIR . '/thirdparty/validate/jquery.validate.pack.js'
  );
  define('SPAN', '<span class="required">*</span>');
}
```

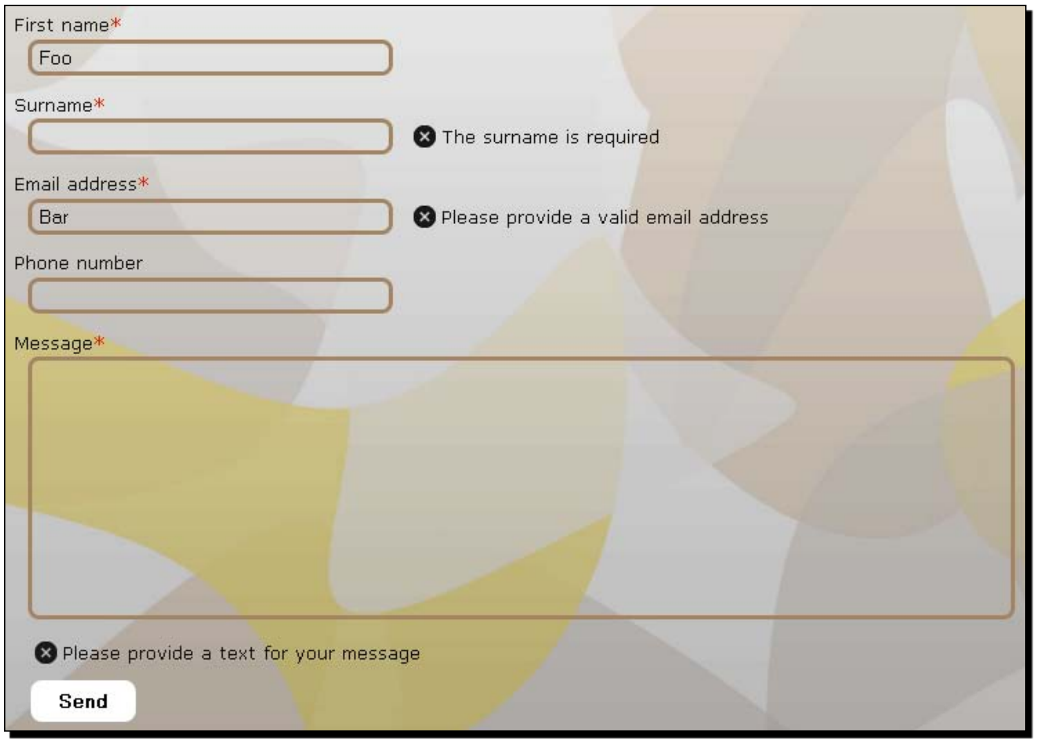
2. In the contact form's `Form()` method, reference the following JavaScript file:

```
Requirements::javascript(
  PROJECT_DIR . '/javascript/ContactPage.js'
);
```

3. Create the file we've just referenced and add the following content to it:

```
$(document).ready(function(){
  $("#Form_Form").validate({
    rules: {
      FirstName: {
        required: true
      },
      Surname: {
        required: true
      },
      Email: {
        required: true,
        email: true
      },
      Phone: {
        digits: true
      },
      Comment: {
        required: true
      }
    },
    messages: {
      FirstName: {
        required: "The first name is required"
      },
      Surname: {
        required: "The surname is required"
      },
      Email: {
        required: "The email address is required",
        email: "Please provide a valid email address"
      },
      Phone: {
        digits: "The phone number must only consist of numbers"
      },
      Comment: {
        required: "Please provide a text for your message"
      }
    }
  });
});
```

4. And that's it! Reload your contact page and try it out. You might see some of the following messages:


First name*

Foo

Surname*

The surname is required

Email address*

Bar

Please provide a valid email address

Phone number

Message*

Please provide a text for your message

What just happened?

First off—why didn't we add the plugin to our theme? The validation is layout independent and we'd need to add the same file to every theme. So it's definitely better to add it in `mysite/thirdparty/` and make it available to all our themes at once.

jQuery's Validation plugin

You need to tell jQuery (already included by `mysite/code/Page.ss`) to use the Validation plugin on our form:

```
$("#Form_Form").validate({ ... });
```

`#Form_Form` is the ID of our form tag. SilverStripe automatically generated that for us, always starting with `Form_` and appending the generating method's name. In our case that's `Form` again, but if we had set up our form through a function `ContactForm()` it would be `#Form_ContactForm`. You can also look it up in your page's HTML source—it would be:

```
<form id="Form_Form" action="contact-us/Form" method="post"
enctype="application/x-www-form-urlencoded">
```

Inside this JavaScript method you have two parts: `rules:` and `messages:` (you can also structure it differently, but this is probably the easier and less repetitive approach).

Rules define what to check, for example that the first name is required:

```
FirstName: {
  required: true
},
```

Messages provide the error message in case the rule isn't met:

```
FirstName: {
      required: "The first name is required"
},
```

But you're not limited to a single rule per field. Both requiring the e-mail field to be filled out and a valid address is just as easy:

```
Email: {
  required: true,
  email: true
},
```

The message for it obviously is:

```
Email: {
  required: "The email address is required",
  email: "Please provide a valid email address"
},
```

For a complete documentation of all the features see `http://docs.jquery.com/Plugins/Validation`.

Pay close attention to commas again

If there is just a single entry or one is the last one of many, don't add a comma after it—some browsers don't take that well. Between multiple entries you must always add a comma. So there's a comma after `rules:` but not after `messages:`, one after `FirstName:` but not after `Comment:`, and so on.

What did this actually change in comparison to the built-in validation?

- We've added a check that a phone number must only consist of numbers.
- The client-side validation is only triggered once the **Send** button is clicked on. Otherwise, it can be a little annoying for the user, but if you want to, you can change it back to SilverStripe's original behavior. See the official documentation.
- We only changed some layout information, but that could have been done with pure CSS.

So in this example it didn't make too much difference, but it will on the booking page coming up next.

Actually, by switching to minified jQuery only, we got rid of quite a lot of JavaScript: Prototype and non-minified jQuery. Altogether we were able to reduce the JavaScript that each visitor needs to download from nearly 250 KB (which is quite bloated) to less than 90 KB. That makes loading pages a bit faster and saves our server some bandwidth, without any changes for our user.

Time for action – using jQuery's Validation plugin on the rent page

For renting a table we'll add some nifty features:

- Tables can only be booked X days in advance. X can be defined in the CMS.
- A table can only be booked for the same day, if it's before 12 a.m. Otherwise, we cannot guarantee that someone really gets the reservation in time and can put out a sign accordingly.
- Reservations can only start as soon as the bar has opened—also definable in the CMS.

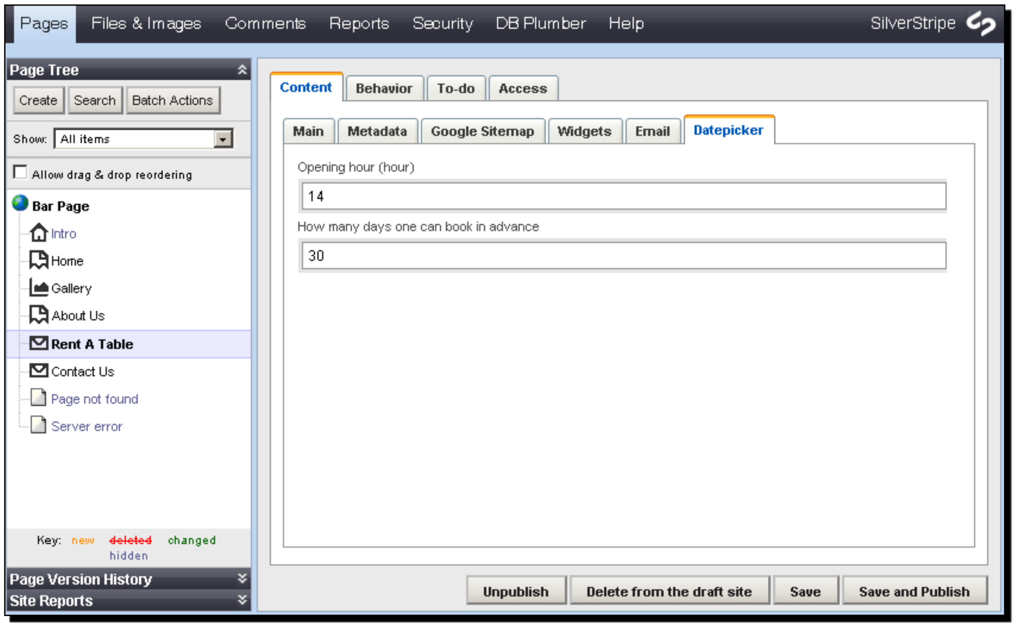
For easy and robust date and time selection, we'll use the Any+Time Datepicker: `http://www.ama3.com/anytime/`. Again it's based on jQuery: pretty simple but very powerful.

So let's do it:

1. In your `RentPage` class (the Model) add:

```
public static $db = array(
  'OpeningHour' => 'Int',
  'ReservationAdvance' => 'Int',
);

public function getCMSFields(){
  $fields = parent::getCMSFields();
  $fields->addFieldToTab(
    'Root.Content.Datepicker',
    new NumericField("OpeningHour", "Opening hour (hour)")
  );
  $fields->addFieldToTab(
    'Root.Content.Datepicker',
    new NumericField(
      "ReservationAdvance",
      "How many days one can book in advance"
    )
  );
  return $fields;
}
```

2. Rebuild the database.
3. You should now be able to add the information in the CMS:



Opening hour (hour)
14
How many days one can book in advance
30

4. Add the following code to the `Form()` method of `RentPage_Controller`:

```
Requirements::css(
  PROJECT_DIR . '/thirdparty/anytime/anytimec.css',
  'screen,projection'
);
Requirements::javascript(
  PROJECT_DIR . '/thirdparty/anytime/anytimec.js'
);

Requirements::javascript(PROJECT_DIR . '/javascript/RentPage.js');
Requirements::customScript('
  $(document).ready(function(){
    var now = new Date();
    var start = new Date(
      now.getFullYear(),
      now.getMonth(),
      now.getDate()' . ((date('H') > 12) ? '+1' : '') . ', ' .
      $this->OpeningHour . ', 00, 00
    );
    var end = new Date(
      now.getFullYear(),
      now.getMonth(),
      now.getDate()+' . $this->ReservationAdvance . ', 23, 59, 59
    );

    $("#Form_Form_Datetime").AnyTime_picker({
      format: "%Y-%m-%e %H:%i",
      firstDOW: 1,
      earliest: start,
      latest: end,
    });
  });
');
```

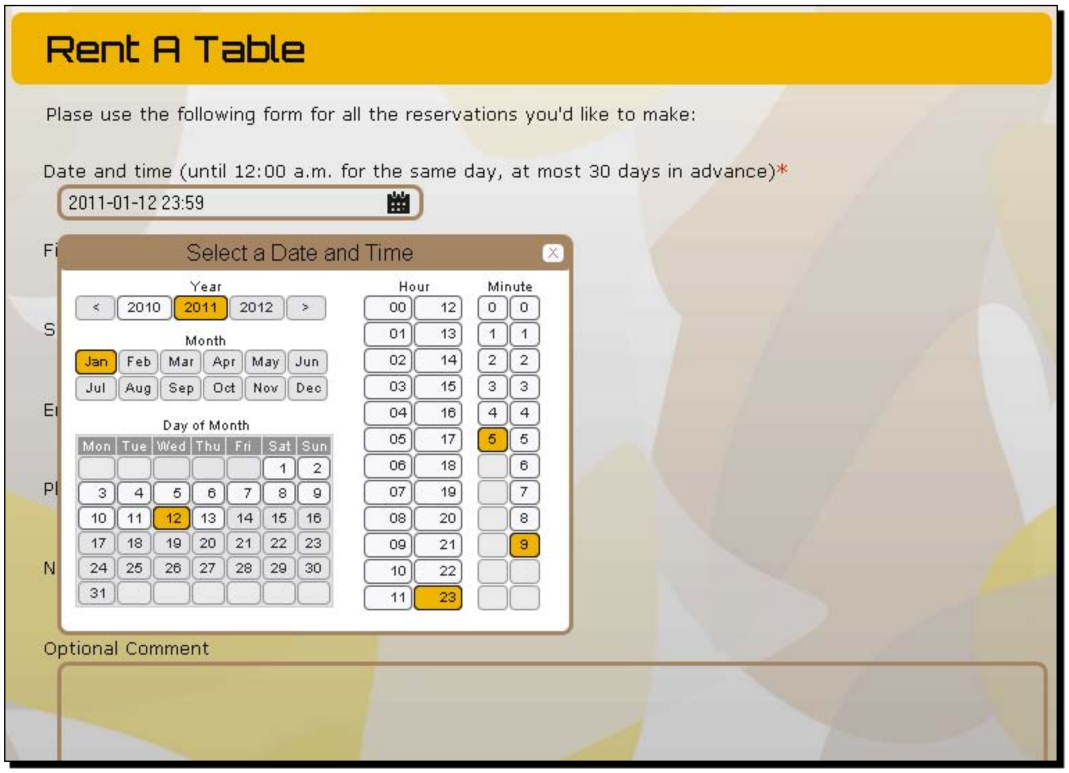
5. Add the following JavaScript to `mysite/javascript/RentPage.js`, much like in the previous example:

```
$(document).ready(function(){
  $("#Form_Form").validate({
    rules: {
      Datetime: {
        required: true
      },
      FirstName: {
        required: true
      },
      Surname: {
        required: true
      },
      Email: {
        required: true,
        email: true
      },
      Phone: {
        required: true,
        digits: true,
        range: [100000, 1000000000000000]
      },
      Count: {
        required: true,
        digits: true,
        range: [1, 100]
      }
    },
    messages: {
      Datetime: {
        required: "Date and time are required"
      },
      FirstName: {
        required: "The first name is required"
      },
      Surname: {
        required: "The surname is required"
      },
      Email: {
        required: "The email address is required",
        email: "Please provide a valid email address"
      },
      Phone: {
        required: "The phone number is required",
        digits: "The phone number must only consist of numbers",
        range: "The phone number must have between 6 and 15
digits"
      },
      Count: {
        required: "The number of people is required",
        digits: "The number of people must only consist of
numbers",
        range: "The number of people must be between 1 and 100"
      }
    }
  });
});
```

6. Add the referenced JavaScript and CSS files, you can find both on `http://www.ama3.com/anytime/`.

7. Remove the last line from the JavaScript file. It always outputs a warning message, preventing anyone from directly linking to the file on the author's server—eating away his bandwidth.

8. Reload the page and marvel at our new toy:



Rent A Table

Plase use the following form for all the reservations you'd like to make:

Date and time (until 12:00 a.m. for the same day, at most 30 days in advance)*

2011-01-12 23:59

Optional Comment

What just happened?

Adding fields to the CMS is routine by now, there shouldn't be anything to describe in detail here. Adding the new JavaScript and CSS files to the form is also old news.

For the Validation plugin we've added one feature: Defining a range of possible values for a number. So when booking a table you can only reserve one or up to one hundred seats—the given numbers are included.

```
Count: {
  required: true,
  digits: true,
  range: [1, 100]
}
```

We can also added minimum and maximum lengths for the name fields with `rangelength: [3, 15]`. But in this example we've decided that it's not worth it—if someone wants to send a phony reservation, we'll simply ignore it. It would also make life unnecessarily difficult for people with exceptionally long or short names.

Setting up the Datepicker

The Datepicker is much more interesting, let's break that down into manageable pieces:

- First we fetch the current timestamp in the variable `now`.

  ```
  var now = new Date();
  ```

- Using that we can set up the possible start time of any reservation—it's the current, year, and month. If it's already past 12 a.m. the reservation can only be made for the next day (`now.getDate()+1`), otherwise for the same day (`now.getDate()`). And it only starts at the time we've set in the CMS.

  ```
  var start = new Date(
    now.getFullYear(),
    now.getMonth(),
    now.getDate()' . ((date('H') > 12) ? '+1' : '') . ', ' .
      $this->OpeningHour . ', 00, 00
  );
  ```

- For the end date it's more or less the same, we're only adding the days bookable in advance to the current day. This handles month and year changes gracefully, so don't worry—you won't get a month with 40 days.

  ```
  var end = new Date(
    now.getFullYear(),
    now.getMonth(),
    now.getDate()+' . $this->ReservationAdvance . ',
    23, 59, 00
  );
  ```

- Now we can put our dates into action. The specific field has the ID of the form (`#Form_Form`), appended with the field's name (`Datetime` in our case). Again, you can simply look it up in the page's HTML source code:

```
$("#Form_Form_Datetime").AnyTime_picker({
```

- This defines the date and time format: `YYYY-MM-DD HH:MM` in our case:

```
  format: "%Y-%m-%e %H:%i",
```

- And we're setting the first day of the week to Monday:

```
  firstDOW: 1,
```

- Finally, we add our `start` and `end` variables to the form:

```
  earliest: start,
  latest: end,
});
```

If you try it out, you'll see that you can really only add dates and times within our predefined time span. And all within our easy-to-use Datepicker. That's it!

Don't worry; server-side validation will come right after this section.

Should you add JavaScript to the Controller?

This is a good question—is the Controller the right place for client-side validation? Actually, we wanted to keep the Controller as separate from the View as possible.

There are two contradictory arguments:

- You should keep all the client-side code in the template or a dedicated JavaScript file. We have easily set up most of our validation code without touching the Controller.
- Server and client-side validation are pretty similar. Although the PHP and JavaScript code are different, their goal will generally be the same. So keeping them as closely together as possible has the advantage that you can easily keep them in sync. Distributing them over multiple files makes this harder.

In our code, we choose the first option for the major part as we don't want to violate MVC too much. With the second approach, you'll also lose any syntax highlighting, as you will be working with strings in PHP, instead of JavaScript within an HTML template, or a dedicated JavaScript file.

Only the JavaScript part relying on database values is located in the Controller. This part should normally sit in the View—where you can also access the database fields. However, as we're using our generic View, which is the same for most pages, we don't really want to include some JavaScript used on a single page only. Therefore, we added this part in the Controller. This is definitely a trade-off—you'll need to make a sensible decision on a case-by-case basis.

Tougher server-side validation

Now that we've tightened up the client-side aspect so much, how can we do the same on the server-side?

Time for action – better server-side validation

Let's do a quick example to check the number of seats that we want to reserve.

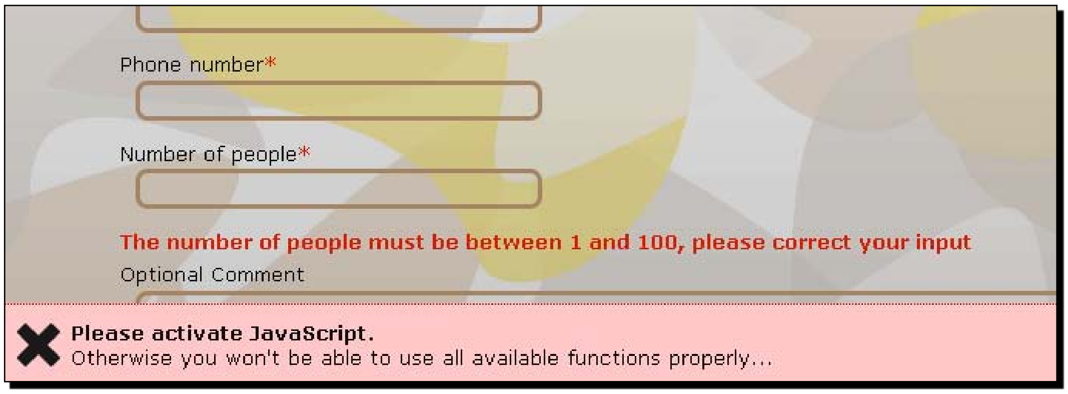
1. Add the following method to the `RentPage_Controller` class:

```
public function sendemail($data, $form){
  if(($data['Count'] < 1) || ($data['Count'] > 100)){
    $form->addErrorMessage(
      'Count',
      'The number of people must be between 1 and 100,
         please correct your input',
      'error'
    );
    Director::redirectBack();
    return;
  }
  Emailer_Controller::sendemail($data, $form);
}
```

2. Disable JavaScript in your browser and reload the page. In case you only entered correct information, your e-mail will be sent just like before.

> In Mozilla Firefox, click on **Tools** and then on **Options**. On the **Content** tab, disable the point **Enable JavaScript**. In Microsoft Internet Explorer, click on the **Tools** drop-down and select **Internet Options**. On the **Security** tab, click on **Custom Level** and **Disable** the point **Scripting**. In Google Chrome, click on **Options**, **Under The Hood**, and then **Content Settings**. Here you will find a **JavaScript** tab under which you can disable it for all pages.

3. But if you want to book zero seats, you'll get the following result:



Phone number*

Number of people*

The number of people must be between 1 and 100, please correct your input

Optional Comment

Please activate JavaScript.
Otherwise you won't be able to use all available functions properly...

The message that JavaScript is disabled isn't a browser feature but is defined via the `<noscript>` tag in the template file `themes/bar/templates/Page.ss`.

Note that all the entered data is being lost in this case. That's why all the fields are empty after being sent back to the form in case of an error. So you should really enable JavaScript for an optimal user experience—just like the message at the bottom says.

What just happened?

Before adding the `sendemail()` method to our `RentPage_Controller`, it will have inherited it from `Email_Controller`. That's still what we want for the `ContactPage_Controller`, but on our rent page, we need to add some server-side validation. So we're overwriting the otherwise inherited method, providing our own implementation.

In case `Count` doesn't have a valid value, we're adding an error message to the form and redirecting back to it (`Director::redirectBack()`). For the `addErrorMessage()` method the arguments are:

- The field for which we want to add the message.
- The message itself.
- The message type—wrapped around the message as a CSS class. By setting the message type you can also mark fields as successfully updated.

In case no error has been detected, we're calling the original `sendemail()` method, passing the same arguments it would have gotten if we hadn't overwritten it. So now it will send the e-mail for us and we don't have to worry about rewriting that functionality.

We could now start validating all of our fields with more specific needs than simply being required. In our case we won't, as we don't really need it. Using client-side validation we've made sure most people won't forget or mistype any important information. In case someone really wants to send manipulated data, we'll simply ignore it.

Although this is appropriate for this requirement, you'll need to assess that for your specific project. There are definitely use cases where this isn't suitable.

Never rely on client-side validation alone, it's just an (important) add-on for server-side validation. JavaScript on the client can be disabled or manipulated easily, so it adds zero protection against attacks and malicious users.

Have a go hero – order a member card

After providing a contact form and being able to book a table, there is one more similar task we need to accomplish. In order to get more contact data and provide a little bonus for returning customers, we're giving out member cards. People having such a card get 10 percent off everything they order.

As we definitely don't want people to fill out paper forms to apply for a member card, we'll provide a similar form on our website. Customers should fill it out online and we'll get the result sent back per mail. The following information is required:

- First name
- Surname
- Date of birth
- Address
- E-mail address
- Phone number

That's very similar to the previous two examples, so that should be easy—right? Try it out.

Instead of mailing the data, we might want to keep it in the database. We'll come back to that requirement in the next chapter. For the moment let's stick to e-mails.

Pop quiz – true or false

Which of the following statements are true, which are false?

- You can select any method name to set up the form, but `Form()` is common
- You should always add JavaScript in the Controller to guarantee security—otherwise it can be changed on the client-side
- You can only trust server-side validation
- You should remove any JavaScript not required on a page

Summary

We learned a lot in this chapter about forms.

Specifically, we covered:

- How to set them up in the Controller
- How to use the data entered into forms
- How SilverStripe handles e-mails
- How to do client-side validation

We've also taken another look at inheritance and how to keep your project DRY. Deciding when to rely on subclasses and when not to is an important consideration when designing easy-to-maintain applications.

Now that we've learned about forms in general, we're ready to do even more with them. In the next chapter, we'll implement search functionality for our page, besides other exciting stuff.

9
Taking Forms a Step Further

We've already built some nice features with forms, however, there's quite a lot more that they can do for us. There are more validation features, fine-grain control over layout changes and we don't have to limit ourselves to sending data as e-mails. We'll save the user supplied input to the database as well.

Specifically we'll:

- Add a simple search engine to our page
- Allow visitors to order a member card—saving the provided information into the database
- Explore how to use server-side code to extend our client-side validation

Sounds good? Then let's get on with it.

Searching our pages

Our custom search engine should allow us to do the following:

- Add a search box in the header section on every page
- Display results on a new page
- Support pagination—if we have more than a given number of results, we want to display them on more than one page

While this may be overkill for our current page, assume that our page might expand in the future. Once we have dozens or even hundreds of pages, searching them will be an important feature.

There is no drop-in module providing the search functionality. But, as always, you can easily roll your own, customized implementation—and that's how you do it. Note that it's based on MySQL's full-text search—if you've opted for another SilverStripe-supported database which uses a different approach, you'll have to try something else.

Time for action – adding search functionality to our page

We'll add a search field to every page. Once some information is entered and sent to the server, the user is taken to a result page. Simple and easy, just what you'd expect.

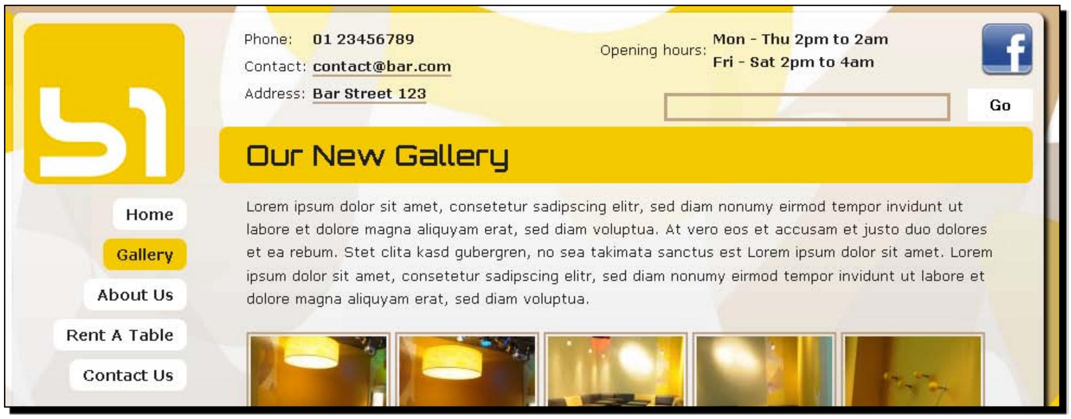
1. First we need to enable the search functionality in `mysite/_config.php` by adding:

```
FulltextSearchable::enable();
```

2. Next we're adding the search form to the header of our page. All we need to do is to add the following placeholder to `Page.ss` and `GalleryPage.ss`. We're intentionally leaving out the intro page as we don't need the search form there:

```
<aside id="top">
  <% include BasicInfo %>
  $SearchForm
</aside>
```

3. After reloading the page and applying some CSS to it, the search box is already operational, but it still looks kind of alien to the rest of our design:

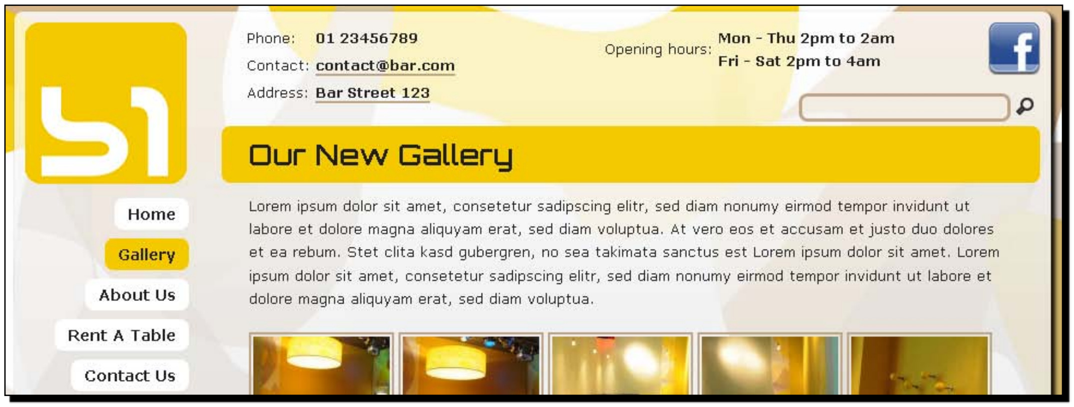


Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

4. While you could amend the search box with a lot more CSS and possibly some tweaking of the underlying HTML with jQuery, there is a much simpler solution. Though the built-in `$SearchForm` placeholder is very convenient for simply adding the search form, it's too limited for us. We can simply replace it with our own form. The original HTML is loaded from the file `sapphire/templates/SearchForm.ss`. We already know how to overwrite that file: create `themes/bar/templates/SearchForm.ss` (not in the `Includes/` or `Layout/` folder) and add the following content to it:

```
<form $FormAttributes class="searchform">
  <input name="Search" type="text" class="searchfield rounded"/>
  <input type="submit" class="searchbutton rounded" value=""/>
</form>
```

5. After applying some styling it now looks like the following image—that's much better:



Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

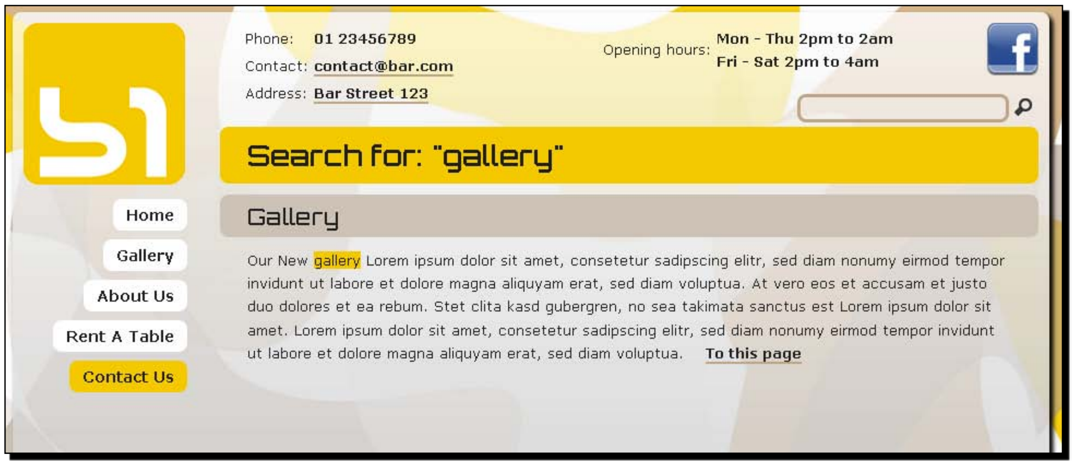
6. However, we still want to customize the logic behind our form. Add the following method to the `Page_Controller` class:

```
public function results($data, $form, $request){
  if(!empty($data['Search'])){
    $templateData = array(
      'Results' => $form->getResults(3, $data),
      'SearchQueryTitle' => $form->getSearchQuery($data),
    );
    return $this->customise($templateData)->renderWith(array(
      'SearchResults',
      'Page',
    ));
  } else {
      Director::redirectBack();
      return;
    }
  }
```

7. Add the result page to display anything found. Do you remember the file `themes/blackcandy/templates/Layout/Page_results.ss` from SilverStripe's default theme, which we mentioned briefly in the second chapter? We'll create our own, customized version of it, but you can of course copy and extend the existing file. Create the page `themes/bar/templates/Layout/Page_results.ss`:

```
<div>
  <img src="$ThemeDir/images/background.jpg" alt="Background"
id="background"/>
</div>

<section id="content" class="transparent rounded shadow">
  <aside id="top">
    <% include BasicInfo %>
    $SearchForm
  </aside>

  <% include Menu %>
  <% include Share %>

  <section class="typography">
    <h1>Search for: &quot;{$SearchQueryTitle}&quot;</h1>
    <% if Results %>
      <% control Results %>
        <article>
          <a class="searchResultHeader" href="$Link">
            <h2>$Title</h2>
          </a>
          <p>
            $Content.ContextSummary &nbsp;&nbsp;&nbsp;
            <a href="$Link">To this page</a>
            <br/>&nbsp;
          </p>
        </article>
      <% end_control %>
      <% if Results.MoreThanOnePage %>
        <div id="PageNumbers">
          <% if Results.NotFirstPage %>
            <a class="prev" href="$Results.PrevLink">
              &larr; Previous page
            </a>
          <% end_if %>
          <span>
          <% control Results.Pages %>
            <% if CurrentBool %>
              &nbsp;&nbsp;<b>$PageNum</b>&nbsp;&nbsp;
            <% else %>
              &nbsp;<a href="$Link">$PageNum</a>&nbsp;
            <% end_if %>
          <% end_control %>
          </span>
          <% if Results.NotLastPage %>
            <a class="next" href="$Results.NextLink">
              Next page &rarr;
            </a>
          <% end_if %>
          <p>
            Page $Results.CurrentPage of $Results.TotalPages
          </p>
        </div>
      <% end_if %>
    <% else %>
      <article>No results found...</article>
    <% end_if %>
  </section>
</section>

<% include Footer %>
```

8. Now you can start our fully customized search, which might look something like this for a single entry:

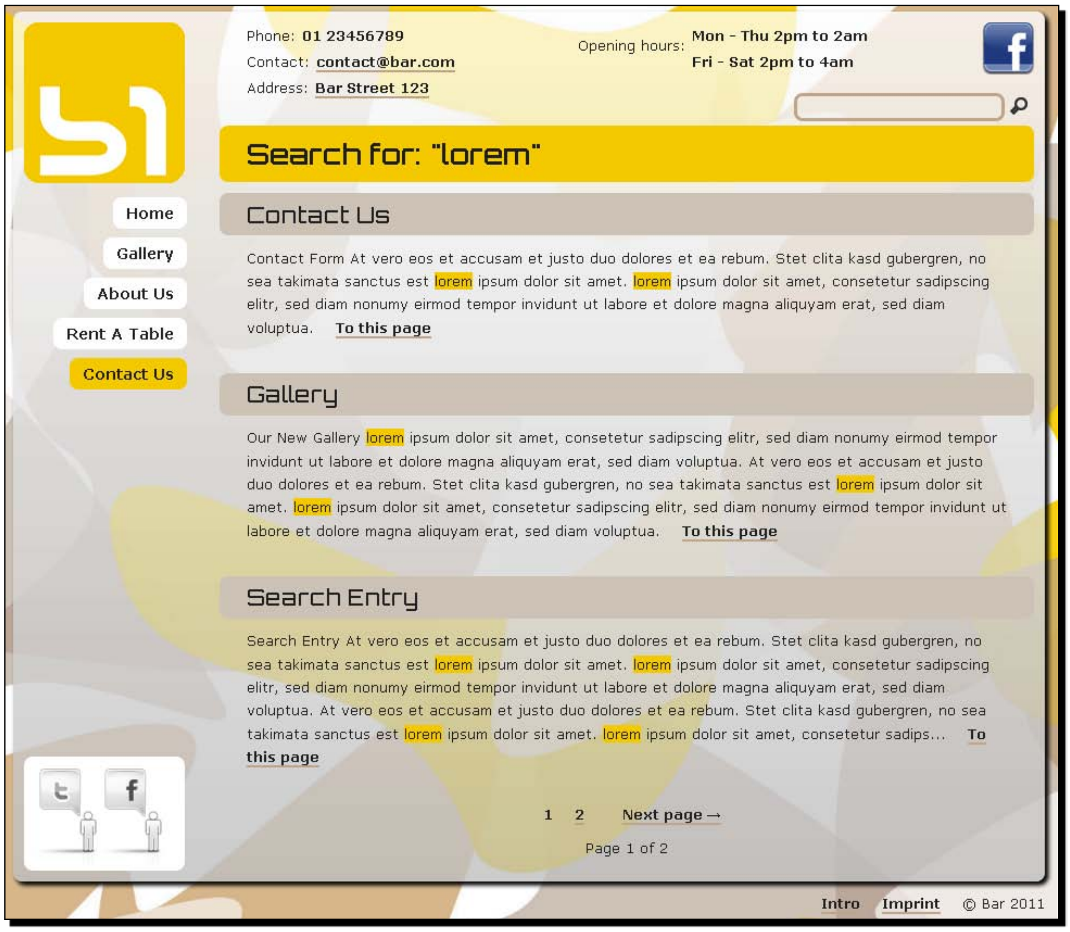


Search for: "gallery"

Gallery

Our New gallery Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. To this page

9. Or like this for multiple items, being split into different pages:



Search for: "lorem"

Contact Us

Contact Form At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est lorem ipsum dolor sit amet. lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. To this page

Gallery

Our New Gallery lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est lorem ipsum dolor sit amet. lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. To this page

Search Entry

Search Entry At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est lorem ipsum dolor sit amet. lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est lorem ipsum dolor sit amet. lorem ipsum dolor sit amet, consetetur sadips... To this page

1 2 Next page →

Page 1 of 2

What just happened?

The first steps are pretty self-explanatory, but where did `$SearchForm` actually come from? It's automagically provided by SilverStripe. But as we're not content with its look we're simply replacing it with plain old HTML. As you can see, you're not required to rely on the HTML Sapphire generates. If you want to build highly customized forms, you can simply build your own. It's quite a lot of work, though, so only do it when absolutely necessary.

`$FormAttributes` is replaced with `enctype="application/x-www-form-urlencoded" method="get" action="/home/SearchForm" id="SearchForm_SearchForm"` (on the home page). You can, of course, replace this with plain HTML as well, but as there's no benefit in doing so, we'll stick with this.

We'll come back to customizing general purpose forms a little later, but for now let's continue with our search form.

The Controller part of the code

First off, why is our function called `results($data, $form, $request)` (`results($data, $form)` would work) while other method names wouldn't work? We're overwriting an already existing method, that's why our search form has been working even before we've added our custom form.

You can find the original in `sapphire/search/ContentControllerSearchExtension.php`. We're basically changing two things: first we're setting the pagination to three entries per page (so that we can see the effect even with only a few results). Second, we're checking for an empty search term and are not processing it but rather redirecting back to the previous page. You can extend it to enforce that the search term must have at least three characters, and so on.

The method `results()` uses the submitted data and the form we just created. `$data['Search']` contains our actual search term. In case it's empty, we're simply redirecting to the original page—it doesn't sound very logical to search for "nothing" and find "everything".

Don't be confused with `$form`—here it's not a form for entering information, but the form used for interacting with the provided search term:

- First, we're creating a named array containing the results of our query. You can add whatever information you deem useful, and the names of the values are up to you.
- Most important is the `Results` value, which contains the actual search results that we fetch via `$form->getResults(3, $data)`. It's a predefined method for searching, the first argument being the number of results per page and the second one our form's result, including the search term.
- Additionally, we're adding the search term itself to the results page—it's always nice to show what the results are actually for.
- Then we're loading the data we just set up (`customise($templateData)`) into the page `Page_results` (`themes/bar/templates/Layout/Page_results.ss`), which itself is embedded in `Page` (`themes/bar/templates/Page.ss`). That's it; the rest is done in the template.

Before diving into the template itself, we should take another look at which templates are chosen when working with methods. We've already looked at the priority of the templates without specific methods being involved; this rule is further added to that system.

Assume you're on the `GalleryPage` type and the current page is rendered with the `results()` method—so the URL could be `/gallery/results`. The priority of templates is:

1. `GalleryPage_results.ss`
2. `GalleryPage.ss`
3. `Page_results.ss`
4. `Page.ss`

Number one is first searched in the `mysite/` folder, secondly in the `themes/` folder, and lastly (if applicable) in any other module directory. Only then number two is taken into account.

So instead of creating both `GalleryPage_results.ss` (in case we're searching on a `GalleryPage` type) and `Page_results.ss` (assuming there are no additional templates for specific page types) we're explicitly calling a specific template. We'll stick to the general naming convention: the most generic page type we want to use is `Page` and the function is `results()`, leaving us with `Page_results.ss`.

And as we want to embed it in the general `Page.ss` we need to reference that as well. Remember the e-mail template? There we've only referenced one file, not using `templates/bar/themes/Page.ss` at all.

The View part of the code

The start of the template `Page_results.ss` should be familiar by now. `$SearchQueryTitle` is the first noteworthy element. That's the element we've defined in the `$templateData` array, and you can access it just like any other placeholder.

Even more powerful is the `Results` element, which is an array itself:

- First we're checking if the array is not empty: `<% if Results %>`.
- Next we're iterating over the results. You get the complete page object, so you can access `$Link`, `$Title`, `$Content`, and so on—just like on any other page.
- `$Content.ContextSummary` is a helper function ideal for displaying search results. It only displays 500 characters and highlights the search term; the HTML output of it is (when searching for "lorem"): `<span class="highlight">lorem</span>`.

- The paging feature is automagically provided by SilverStripe. Most of the features are self-explanatory: `MoreThanOnePage`, `NotFirstPage`, `NotLastPage`, `PrevLink`, and `NextLink`.
- With `<% control Results.Pages %>` you can loop over all the available pages in the paging. `%CurrentBool` checks if you are currently on this page, `$PageNum` holds the page number of the current iteration, and `$Link` provides the URL.

And that's everything you need to know for building a personalized, fully-customized search function for your site.

Advanced options

There are cases where this search function is not enough. You can further limit SilverStripe's search to specific content, sort by relevancy, and so on with the help of more powerful modules such as `http://silverstripe.org/sphinx-module/`. It integrates `http://sphinxsearch.com` into your page's search.

Customizing forms even further

After customizing our search form, let's look into how this can be done for general-purpose forms. Perhaps you might want to customize your form's appearance—providing some additional descriptions, tooltip information, and so on. Although you can achieve a lot of this in the Controller (we've added custom CSS classes for example), don't overdo it as you shouldn't do the View's job in the Controller.

Overwriting the global template

By default, forms are rendered via `sapphire/templates/Includes/Form.ss`:

```
<% if IncludeFormTag %>
<form $FormAttributes>
<% end_if %>
  <% if Message %>
  <p id="{$FormName}_error" class="message $MessageType">$Message</p>
  <% else %>
  <p id="{$FormName}_error" class="message $MessageType"
style="display: none"></p>
  <% end_if %>

  <fieldset>
    <% if Legend %><legend>$Legend</legend><% end_if %>
    <% control Fields %>
```

```
        $FieldHolder
      <% end_control %>
      <div class="clear"><!-- --></div>
    </fieldset>

    <% if Actions %>
    <div class="Actions">
      <% control Actions %>
        $Field
      <% end_control %>
    </div>
    <% end_if %>
  <% if IncludeFormTag %>
  </form>
  <% end_if %>
```

If you want to add or remove fieldsets or customize CSS classes for all of your forms, copy the file to `themes/bar/templates/Includes/Form.ss` and style away. The elements you can use are:

- `$IncludeFormTag`: Whether or not you want to automatically insert the `<form>` tags or rather write your own. In our examples, we'll never add them manually, so you can simply remove the if statement.
- `$FormAttributes`: We've already covered this element; most importantly it defines the URL under which the form should be processed.
- `$FormName`: The name of the method creating the form.
- `$Message`: The second argument of `addErrorMessage()`.
- `$MessageType`: The third argument of `addErrorMessage()`.

As a reminder, we used `addErrorMessage()` in the previous chapter. The first argument is the field in which the error happened, for example `FirstName`. The second argument is the error message displayed to the user and the third (optional) one is the CSS class applied to it.

- `$Legend`: The description of the form. Instead of doing `return new Form()`, use the following code to add the legend:

  ```
  $form = new Form($this, "Form", $fields, $actions);
  $form->setLegend("The legend...");
  return $form
  ```

- `$Fields`: All the fields in the form. Basically, everything where you can input some kind of information (text, textarea, checkbox, and so on).

- `$FieldHolder`: Represents each field in the HTML form.
- `$Actions`: All the actions in the form—basically all the buttons.
- `$Field`: When used inside `<% control Actions %>`, it iterates over all the available form actions.

If you need more fine-grained control and you can afford to invest more time, take a look at the next approach.

Handcrafting forms with SilverStripe

Assume you've set up your form in the method `BaseForm()` in the Controller with the fields `Title`, `Abstract`, `Longtext`, and so on. You can build a handcrafted template for that, in the template file where you want to display the form, as follows—note that we're reusing some of the elements from the previous section:

```
<% control BaseForm %>
  <form $FormAttributes>
    <% if Message %>
      <p id="{$FormName}_error" class="message $MessageType">
        $Message
      </p>
    <% else %>
      <p id="{$FormName}_error" class="message $MessageType"></p>
    <% end_if %>

    <h3>Basic Information</h3>
    <div class="formsection">
      <div class="label">
        <label for="Form_BaseForm_Title">
          Project Title<span class="required">*</span>
        </label>
      </div>
      <div class="field">
        $dataFieldByName(Title)<br/>
        <span class="$dataFieldByName(Title).MessageType">
          $dataFieldByName(Title).Message
        </span>
      </div>
      <div class="label">
        <div class="infotext">
          Short description - 400 to 800 characters long
        </div>
        <label for="Form_BasicForm_Abstract" id="counter">
          Abstract<span class="required">*</span>
        </label>
```

```
          </div>
          <div class="field">
            $dataFieldByName(Abstract)<br>
            <span class="$dataFieldByName(Abstract).MessageType">
              $dataFieldByName(Abstract).Message
            </span>
          </div>
        </div>

        <h3>Detailed Information</h3>
        <div class="formsection">
          <div class="label">
            <label for="Form_BaseForm_Longtext">
              ...
          </div>
        </div>

        $dataFieldByName(SecurityID)
        <% if Actions %>
          <div class="Actions">
            <% control Actions %>$Field<% end_control %>
          </div>
        <% end_if %>
      </form>
    <% end_control %>
```

The placeholder `$dataFieldByName(Title)` fetches the `Title` field—which would be one of the `$FieldHolder` elements from the previous example. By appending `.Message` or `.MessageType` to it, you can fetch error messages for this specific field and the type of the error.

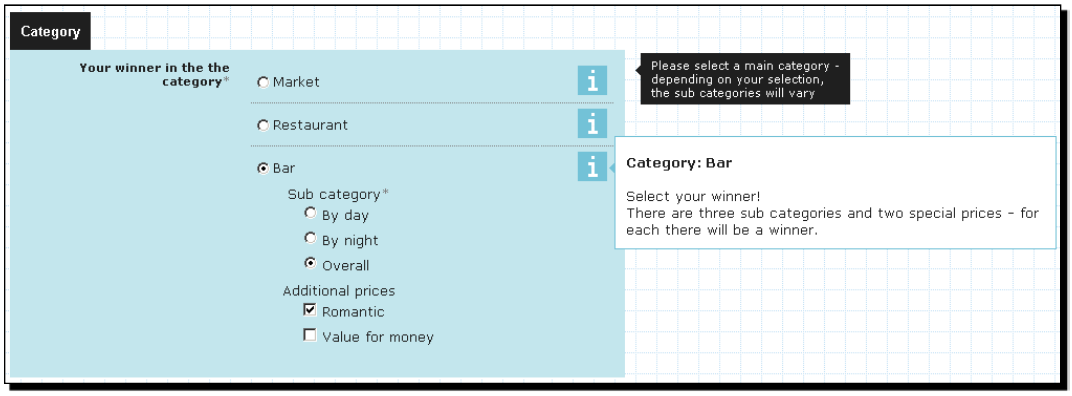
`$SecurityID` inserts a security token to protect you against cross-site request forgery attacks (CSRF). The output in the HTML source code will look something like this:

```
<input class="hidden" type="hidden" id="Form_Form_SecurityID"
name="SecurityID" value="c194ea1fecf0708575d45853371e59525181ef50" />
```

The server inserts this token when generating a form and only accepts a response that includes the same token. This prevents CSRF attacks where a malicious website issues unauthorized commands to another website in the name of the current user. Actually, this exploits the trust a site has in a user's browser.

Even though this is not vital for our simple forms (there isn't really any sensitive information involved), you should still always make use of such security mechanisms. Better to be safe than sorry!

As you can see in our example with `<div class="infotext">`, you can insert any markup—providing additional helper messages, info popups, and much more. See the following example for a form created with this approach—featuring additional static information boxes (gray box), info popups if you hover over the "i" button (white box with blue border), and more:



Please select a main category - depending on your selection, the sub categories will vary

Category: Bar

Select your winner!
There are three sub categories and two special prices - for each there will be a winner.

Although this design approach gives us complete control over the form's look, it also requires a lot of additional work. You'll need to find the right balance for your requirements.

Saving data to the database

Have you built the member card example in the previous chapter's "Have a go hero" section? You should as we're now extending it. If not, don't worry as you can build it now while we go along.

Time for action – extending the member card form

In the previous chapter we've sent the data via e-mail. However, it would be very useful to store it in the database as well—to have a complete and easily searchable history.

So let's start hacking the previous member card example:

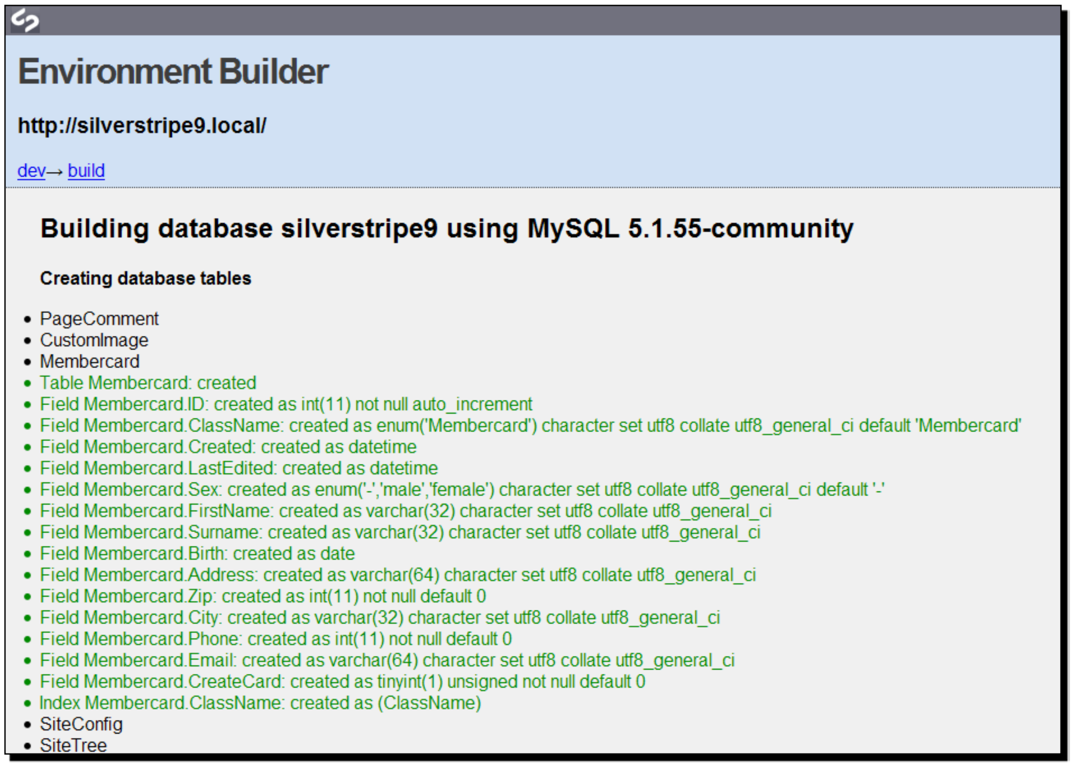
1. In the e-mail template `Email.ss` add the new fields (highlighted in our example) we'll need later on:

```
<ul>
  <% if Sex %><li>Sex: $Sex</li><% end_if %>
  <% if Datetime %><li>Date and time: $Datetime</li><% end_if %>
  <% if FirstName %><li>First name: $FirstName</li><% end_if %>
  <% if Surname %><li>Surname: $Surname</li><% end_if %>
  <% if Birth %><li>Date of birth: $Birth</li><% end_if %>
  <% if Address %><li>Address: $Address</li><% end_if %>
  <% if Zip %><li>Zip code: $Zip</li><% end_if %>
  <% if City %><li>City: $City</li><% end_if %>
  <% if Email %><li>Email address: $Email</li><% end_if %>
  <% if Phone %><li>Phone number: $Phone</li><% end_if %>
  <% if Count %><li>Number of people: $Count</li><% end_if %>
  <% if Comment %><li>Message: $Comment</li><% end_if %>
</ul>
```

2. Next, we need to define a separate Model for our data, `Membercard` sounds like a good choice:

```
class Membercard extends DataObject {

  public static $db = array(
    'Sex' => "Enum('-, male, female', '-')",
    'FirstName' => 'Varchar(32)',
    'Surname' => 'Varchar(32)',
    'Birth' => 'Date',
    'Address' => 'Varchar(64)',
    'Zip' => 'Int',
    'City' => 'Varchar(32)',
    'Phone' => 'Int',
    'Email' => 'Varchar(64)',
    'CreateCard' => 'Boolean',
  );

  public static $indexes = array(
    'Email' => true,
  );


}
```

3. Rebuilding the database will add the table just as we specified:



Environment Builder

http://silverstripe9.local/

dev→ build

Building database silverstripe9 using MySQL 5.1.55-community

Creating database tables

```
• PageComment
• CustomImage
• Membercard
• Table Membercard: created
• Field Membercard.ID: created as int(11) not null auto_increment
• Field Membercard.ClassName: created as enum('Membercard') character set utf8 collate utf8_general_ci default 'Membercard'
• Field Membercard.Created: created as datetime
• Field Membercard.LastEdited: created as datetime
• Field Membercard.Sex: created as enum('-','male','female') character set utf8 collate utf8_general_ci default '-'
• Field Membercard.FirstName: created as varchar(32) character set utf8 collate utf8_general_ci
• Field Membercard.Surname: created as varchar(32) character set utf8 collate utf8_general_ci
• Field Membercard.Birth: created as date
• Field Membercard.Address: created as varchar(64) character set utf8 collate utf8_general_ci
• Field Membercard.Zip: created as int(11) not null default 0
• Field Membercard.City: created as varchar(32) character set utf8 collate utf8_general_ci
• Field Membercard.Phone: created as int(11) not null default 0
• Field Membercard.Email: created as varchar(64) character set utf8 collate utf8_general_ci
• Field Membercard.CreateCard: created as tinyint(1) unsigned not null default 0
• Index Membercard.ClassName: created as (ClassName)
• SiteConfig
• SiteTree
```

4. In the `MembercardPage.php` create an empty Model—there's nothing new to specify here.

5. Again the Controller is where the action happens, starting off with the general setup and jQuery form validator. We're leaving out the page-specific `MembercardPage.js` as we've already covered all of its features in the previous chapter:

```
class MembercardPage_Controller extends Emailer_Controller {

  protected $subject = "Membercard Bar";

  protected function Form(){
    Requirements::css(
      PROJECT_DIR . '/thirdparty/anytime/anytimec.css',
      'screen,projection'
    );
    Requirements::javascript(
      PROJECT_DIR . '/thirdparty/anytime/anytimec.js'
    );
    Requirements::javascript(
      PROJECT_DIR . '/javascript/MembercardPage.js'
    );
```

6. Next we're setting up the form fields in the `Form()` method (we've omitted the CSS classes for clarity, but you know how to do those):

```
  $sex = new OptionsetField(
    'Sex',
    'Sex' . SPAN,
    array(
      'male' => 'Male',
      'female' => 'Female',
    ),
    ''
  );
  $firstName = new TextField(
    'FirstName',
    'First name' . SPAN,
    '',
    32
  );
  $surname = new TextField(
    'Surname',
    'Surname' . SPAN ,
    '',
    32);
  $birth = new DateField(
    'Birth',
    'Date of birth<noscript><br/>Format: YYYY-MM-DD</noscript>' .
        SPAN
  );
  $birth->setConfig('dateformat', 'yyyy-MM-dd');
  $birth->setConfig('min', '-100 years');
  $birth->setConfig('max', '-16 years');
  $address = new TextField(
    'Address',
    'Address' . SPAN ,
    '',
    64
  );
  $zip = new NumericField(
    'Zip',
    'Zip code' . SPAN
```

```
);
$city = new TextField(
  'City',
  'City' . SPAN,
  '',
  32
);
$email = new EmailField(
  'Email',
  'Email address' . SPAN,
  '',
  32
);
$phone = new NumericField(
  'Phone',
  'Phone number' . SPAN
);
$accept = new CheckboxField(
  'Accept',
  'Your data may be digitally processed and ' .
      we may inform you about the latest news per mail' . SPAN
);
```

7. Still in the `Form()` method we'll now put together all the fields we've just created, and add the form action and validator. Once this is done we will finally return the finished form object:

```
$fields = new FieldSet(
  $sex,
  $firstName,
  $surname,
  $birth,
  $address,
  $zip,
  $city,
  $email,
  $phone,
  $accept
);
$send = new FormAction('sendemail', 'Send');
$actions = new FieldSet(
  $send
);
$validator = new RequiredFields(
  'Sex',
        'FirstName',
        'Surname',
        'Birth',
        'Address',
        'Zip',
        'City',
        'Email',
        'Phone',
        'Accept'
      );
      $form = new Form($this, 'Form', $fields, $actions, $validator);
      return $form;
    }
```

8. Define the logic for processing the form, starting off with the server-side validation:

```
public function sendemail($data, $form){
  $error = false;
  if(strlen($data['Zip']) < 4){
    $form->addErrorMessage(
      'Zip',
      'The zip code must at least have four digits',
      'error'
    );
    $error = true;
  }
  if(strlen($data['Zip']) > 5){
    $form->addErrorMessage(
      'Zip',
      'The zip code must at most have five digits',
      'error'
    );
    $error = true;
  }
  if(strlen($data['Phone']) < 6){
    $form->addErrorMessage(
      'Phone',
      'The phone number must at least have 6 digits',
      'error'
    );
    $error = true;
  }
  if(strlen($data['Phone']) > 15){
    $form->addErrorMessage(
      'Phone',
        'The phone number must at most have 15 digits',
        'error'
      );
      $error = true;
    }
    if($this->duplicateEmail($data['Email'])){
      $form->addErrorMessage(
        'Email',
        'The given email address <b>' . $data['Email'] .
            '</b> already exists', 'error'
      );
      $error = true;
    }
    if($error){
      Director::redirectBack();
      return;
    }
```

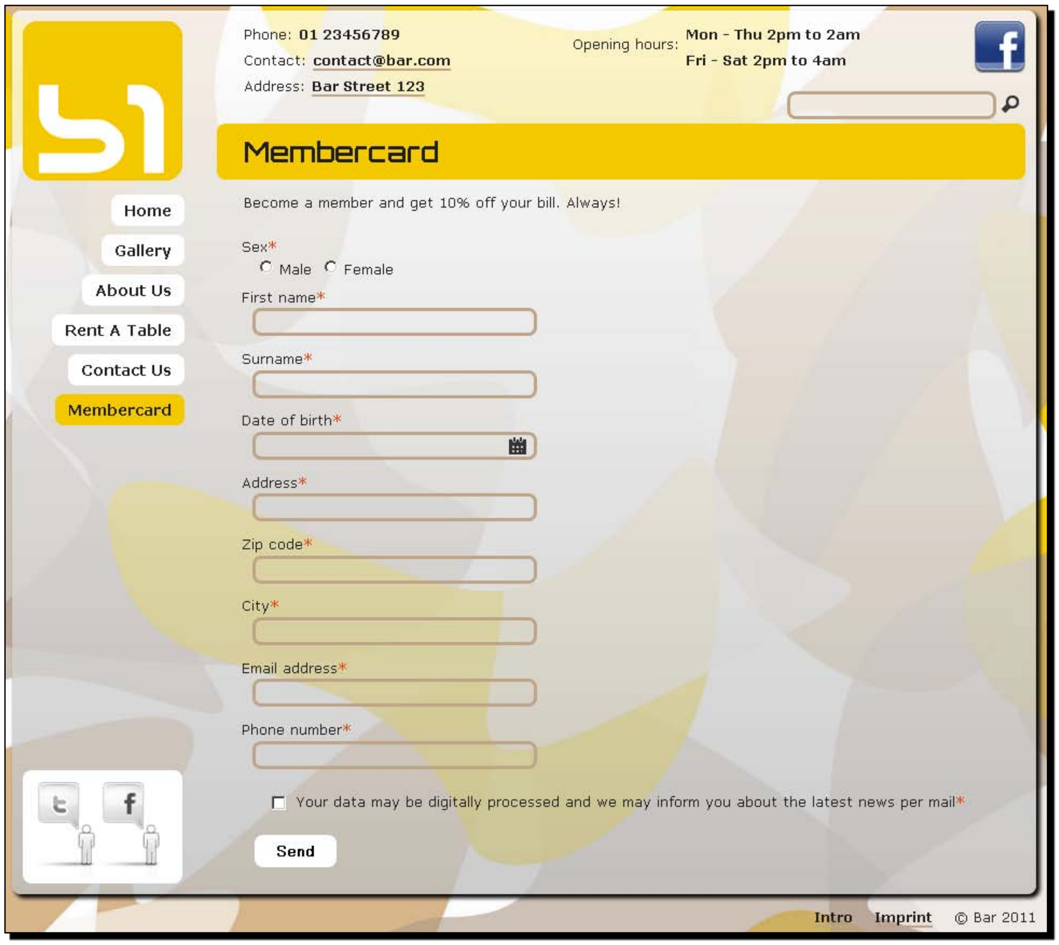
9. Continuing in the `sendemail()` method, we're now saving the database and sending the e-mail:

```
    $membercard = new Membercard();
    $fields = array(
      'Sex',
      'FirstName',
      'Surname',
      'Birth',
      'Address',
      'Zip',
      'City',
      'Email',
      'Phone',
    );
    $form->saveInto($membercard, $fields);
    $membercard->CreateCard = true;
    $membercard->Birth = $data['Birth'];
    $membercard->write();
    Emailer_Controller::sendemail($data, $form);
  }
```

10. Finally, we need to add the method `duplicateEmail()`, which we've already called in the server-side validation part:

```
  protected function duplicateEmail($email){
    $SQL_email = Convert::raw2sql($email);
    $membercard = DataObject::get_one(
      'Membercard',
      "\"Email\" = '$SQL_email'"
    );
    if($membercard){
      return true;
    } else {
      return false;
    }
  }
```

11. Reload the page, your new form should look like this (with some minor styling added):



Membercard

Become a member and get 10% off your bill. Always!

Sex*
Male Female

First name*

Surname*

Date of birth*

Address*

Zip code*

City*

Email address*

Phone number*

Your data may be digitally processed and we may inform you about the latest news per mail*

What just happened?

We haven't actually introduced anything new in the e-mail template, so let's start off with the `DataObject` we created. This is the Model part of our code, which defines our data structure so that we can save it to the database.

The Model part of the code

`DataObjects` are also nothing new and we've already discussed all the fields used here in Chapter 4. Additionally, it should be obvious why we can't add the fields to the member page class—if not take a look at Chapter 7 again. But why are we not only providing "male" and "female" as values in the `Sex` field? We'll come to that a little later.

The `CreateCard` field is for flagging cards we still need to manufacture. As soon as we've done that, we'll flip the value, helping us keep track of the cards we still need to produce.

Finally we're adding `$indexes`. This array defines on which attributes database indexes should be created—greatly enhancing the performance for operations on these fields. As we're checking the uniqueness of the `Email` field, it's definitely a good idea to add such an index for it.

Setting up the fields

Setting up the JavaScript validator for the form is already second nature for us, so let's look at the new elements.

New arguments for TextField

You might have noticed that we've added two arguments to some `TextField()` elements and `EmailField()` (a subclass of `TextField()`). Why?

Take a look at the definition of the `FirstName` field—it is a `Varchar(32)`. So it can hold a maximum of 32 characters. What happens if you try to enter more characters into the text field? Nothing obvious - the input is accepted, but only the first 32 are saved in the database. Now that's pretty nasty, isn't it?

In order to avoid that we're adding a limit—you simply won't be able to type in more characters than the field can store. Simply add two arguments to `TextField()`. The third one is the optional default value (empty in our example, as users should fill in their information) and the fourth one defining the maximum number of allowed characters. This adds the `maxlength` attribute in the HTML's form definition.

That may not be as fancy as a JavaScript validation, but should avoid any accidental problems and is dead simple to set up.

> This won't catch any form manipulation as no server-side validation is applied. However, no harm can be done, as strings that are too long are simply cut off. So regular users can't accidentally run into problems and attackers are not able to exploit this.

OptionsetField

The `OptionsetField()` is the first really new thing in our code. Its parameters are:

- The name of the field used.
- The label—just like all the other fields.
- The available arguments up for selection. You can either add them manually via an associative array or fetch them from the database. As we won't add the "-" option and wanted to capitalize the labels, we went with the manual options.

  In order to fetch the values from the database you'd only need to replace the array with:

  ```
  singleton('Membercard')->dbObject('Sex')->enumValues()
  ```

> Singleton is a well-known design pattern. It will always return an instance of the same basic object. In case no such object exists, it is created; later on it's simply reused—perfect for this use case as we only need it for retrieving the possible values.

- The last (optional) argument defines the default value. We've only added it for verbosity, simply leaving it out will have the same effect as providing an empty one: nothing is selected.

CheckboxField

The use of `CheckboxField()` is pretty obvious. However, note that we're only using this field for rendering the form, not when processing and storing the information. By using the validator we're making sure people accept our terms, but that's it.

> You need to check if this is legally sufficient in your region, especially when sending regular newsletters as the restrictions can sometimes be pretty tough.

DateField options

In the previous example we've kept the date and time field simple as it was only mailed out. Here we're saving it to the database, so let's create a real date instead of a text field.

The arguments when creating a `DateField()` instead of a `TextField()` are the same, but you have some additional options:

- The date format to be used—we're sticking to the format from the previous chapter: `yyyy-MM-dd`, for example `2011-06-25`. Note that you should lowercase year and day, whereas month should be uppercased. Using `mm` will have months go from `00` to `11` instead of `01` to `12`. This doesn't follow PHP's default configuration, but is SilverStripe-specific.
- The minimum and maximum value—as we've disabled the default client-side validation, this will only touch the server-side. You can either provide a date (in the format you've defined) or set a difference to the current date. For example, "1 day" for tomorrow, "-5 days" for five days ago, and so on.

Checking for invalid data and duplicate e-mail addresses

As we've already said multiple times: server-side validation is crucial! Here we're checking the minimum and maximum length of zip codes and phone numbers. In case they are not met, users are redirected to the previous page and an error message is shown.

There is a method `CustomRequiredFields()`, which does the same as `RequiredFields()`, but can be extended with custom, server-side validation rules. However, it shouldn't be used any more—if you find it in an application, better replace it with the method we've used in our form.

Before adding a user to our database we need to check if they already exist. The easiest approach is by using the e-mail address—ignoring the fact that people can use multiple addresses. This is not 100 per cent foolproof, but should be good enough.

For easier checking, we create a custom method `duplicateEmail()`. Make sure it's `protected`, so it can't be called directly from the website like `public` methods.

First, we're converting the given e-mail address to an SQL safe string, using `raw2sql()`. Always do that when interacting with user inputs—never trust them! There are also a `raw2xml()` and `raw2js()` in case you want to output some user-provided input—for a complete list, locate the `Convert` class in `/sapphire/core/`.

With our e-mail string sanitized, we next try to fetch a single member card object with the same address: `DataObject::get_one('Membercard', "\"Email\" = '$SQL_email'")`. If such an object exists, we're returning `true`, otherwise `false`.

Double quote field names

As shown in our example, field names should be double-quoted. This helps SQL queries to stay database independent so they can be used on MySQL as well as PostgreSQL, MS SQL, SQLite, and Oracle. And don't use back ticks—they are MySQL specific.

Saving a new entry to the database

Now that all the checks have been made, we can finally save the data. To do this, we first create a new, empty object: `$membercard = new Membercard()`.

Then we load the form data into this object: `$form->saveInto($membercard, $fields)`. The second, optional argument defines which values are actually taken into consideration.

This might seem superfluous, but in fact it isn't. The method `saveInto()` tries to add any sent form data into the database. If someone manipulates this data and names fields which are not publicly accessible, they would be happily written to the database. In our example, that's not too much of a problem; they could only set the `CreateCard` to true. However, in other cases you might have a credit field (the given user has a credit of 100€ for example), inaccessible to the frontend users. If you don't specify the fields and someone guesses the field name, you'd have a major problem if someone set it to 10,000€. So always use the `$fields` argument—better be safe than sorry.

Next we set the `CreateCard` value to `true`—by default all cards still need to be created. Explicitly setting the `Birth` value is done because sometimes the date format is not correctly applied. Manually setting the value here corrects the problem though.

Finally, we need to save the object. So far it has only existed in memory. After that we can call the parent function, sending out the e-mail.

Once you've saved at least one user in the database, you can see and edit them with the help of our good friend DB Plumber:

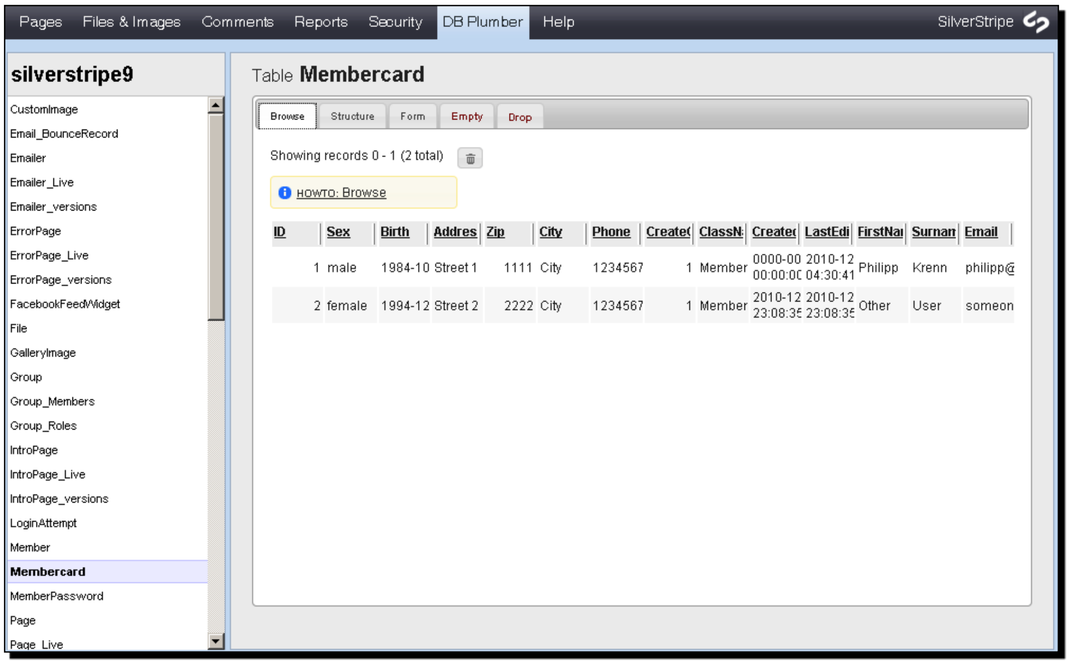


Table **Membercard**

Showing records 0 - 1 (2 total)

ID	Sex	Birth	Addres	Zip	City	Phone	Create(	ClassN:	Createc	LastEdi	FirstNa	Surnam	Email
1	male	1984-10	Street 1	1111	City	1234567	1	Member	0000-00 00:00:0C	2010-12 04:30:41	Philipp	Krenn	philipp@
2	female	1994-12	Street 2	2222	City	1234567	1	Member	2010-12 23:08:35	2010-12 23:08:35	Other	User	someon

And that's our basic member card form—quite a piece of work!

Field types overview

So far we've covered the following field types:

- `CheckboxField()`
- `DateField()`
- `EmailField()`
- `HtmlEditorField()` in the CMS, but it's used just like a `TextField()`
- `NumericField()`
- `OptionsetField()`
- `TextareaField()`
- `TextField()`

Some fields are very similar to those and you'll already know how to use them:

- `CurrencyField()` is for the data type Currency, used just like a regular `TextField()` but including additional validators.
- `DropdownField()` is a super class of `OptionsetField()`. It's instantiated in the same way and is then rendered as a drop-down menu.
- `DatetimeField()` and `TimeField()` are similar to `DateField()`. You can define the format via `->setConfig('datavalueformat' => 'Y-M-d H:m:s')` (for date and time), but you don't have the ability to set minimum and maximum values.

And then there are some more fields we should probably take a quick look at.

ConfirmedPasswordField

This is the minimal version, adding two input fields—"Password" and "Confirm Password":

```
new ConfirmedPasswordField(
  'Password'
)
```

The input in these fields is displayed as asterisks and it's validated if the values are equal. You can optionally add a label as a second argument.

HiddenField

There are situations when you need to add some invisible information to a form, so that it's sent back when the form is submitted—for tracking user activity, for example—but don't forget to validate the value as it could have been manipulated.

```
new HiddenField(
  "HiddenField",
  "(Optional) invisible title",
  "(Optional) value"
)
```

More

There are many more fields for things like handling relationships between classes.
But we'll leave it at that, as it's enough to get you started.

In case you need more, you should definitely take a look at the API documentation, for example at `http://api.silverstripe.org/2.4/li_forms.html`. Once you're there, you have all the available form fields on the left-hand side. Do some digging around and you'll be a form champion in no time.

Checking the e-mail address for uniqueness

We've already validated the uniqueness of e-mail addresses on the server-side. But having people submit the form and only then getting back the error message isn't the most user-friendly solution. Why not provide a client-side solution as well?

Time for action – checking the e-mail's uniqueness with Ajax

Using the tools already in place, this is far simpler than you'd expect:

1. Add the following method to the `MembercardPage_Controller` class:

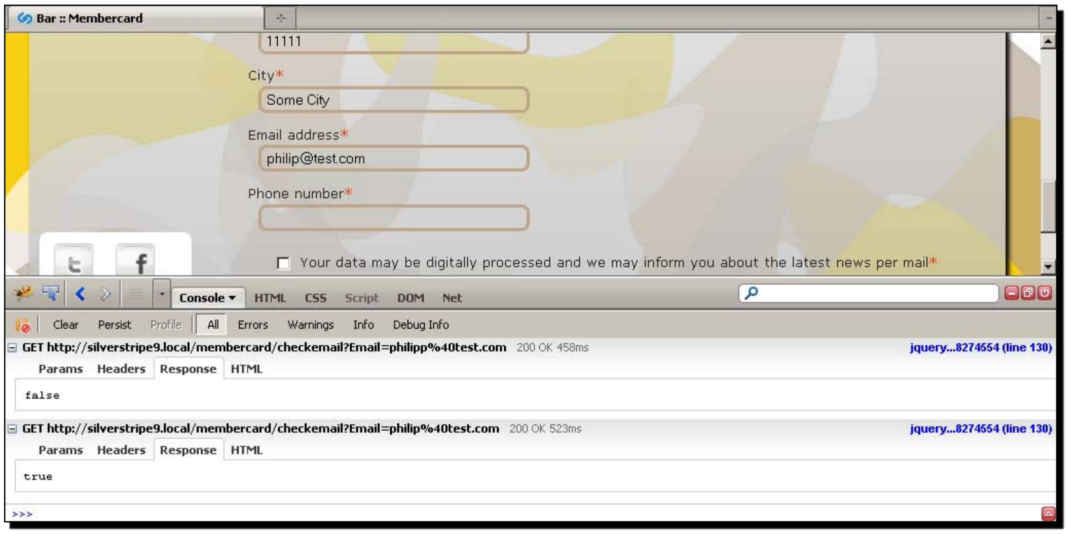
```
public function checkemail(){
  if($this->duplicateEmail($this->request->getVar('Email'))){
    echo 'false';
  } else {
    echo 'true';
  }
}
```

2. In the custom JavaScript validator in `mysite/javascript/MembercardPage.js`, make sure you have at least the following code. Only the highlighted line is really new:

```
$(document).ready(function(){
  $("#Form_Form").validate({
    rules: {
      Email: {
        required: true,
        email: true,
        remote: location.pathname + "checkemail"
      },
    },
    messages: {
      Email: {
        required: "The email address is required",
        email: "Please provide a valid email address",
        remote: "The given email address already exists"
      }
    }
  });
});
```

3. After reloading the page, the uniqueness of the e-mail address will be validated as soon as you've left the field—before submitting the form.

4. In the following image we've first tried to validate an already existing e-mail address (`philipp@test.com`) and then a new one (`philip@test.com`). With the help of Firefox's Firebug plugin (which we used earlier for our quick performance test) we can see what's actually going on. In the screenshot's bottom part we can see the Ajax requests made for checking the e-mail address. Entering an already existing address returns `false` whereas a new (unique) e-mail address returns `true`:


Bar :: Membercard
11111
City*
Some City
Email address*
philip@test.com
Phone number*
Your data may be digitally processed and we may inform you about the latest news per mail*
Console HTML CSS Script DOM Net
Clear Persist Profile All Errors Warnings Info Debug Info
Params Headers Response HTML

```
GET http://silverstripe9.local/membercard/checkemail?Email=philipp%40test.com  200 OK 458ms   jquery...8274554 (line 130)
false
GET http://silverstripe9.local/membercard/checkemail?Email=philip%40test.com  200 OK 523ms   jquery...8274554 (line 130)
true
```

What just happened?

First we're setting up a new public method `checkemail()`. Assuming your member card page has the URL `/membercard`, your method will be available under `/membercard/checkemail`. We're attaching the URL parameter `Email` with the value of the given e-mail address to it, resulting in `/membercard/checkemail?Email=givenEmail`.

In the function itself we're taking the URL parameter (`$this->request->getVar('Email')`) and checking it via the already available method `duplicateEmail()`.

Instead of `$this->request->getVar('Email')`, you could have also used `$_GET['Email']`. The later one is the regular PHP version whereas the first one is a SilverStripe-specific handler. In general, they both do exactly the same thing (neither provide any form of input validation). Still it's advisable to use the SilverStripe-specific one as it may avoid problems when requests are done programmatically, for example, when doing automated tests or executing bulk actions. Although this isn't a concern in our example, you should probably get used to this way of handling it, probably saving yourself unnecessary trouble later on.

Depending on the outcome, we're returning the string `true` for a unique and `false` for a duplicate e-mail address.

Due to PHP's dynamic typing you can't simply return the Boolean `true` or `false`. They would automatically be converted to the string `1` and an empty string respectively. As we need the strings `true` and `false`, we're returning them instead.

In jQuery's validator we're simply stating that the value should be validated against the method `checkemail` on the current page. The JavaScript code `location.pathname + "checkemail"` would lead to the URL `/membercard/checkemail` (sticking to the previously chosen page address). You can, of course, also use an absolute URL like `http://check.com/membercard/checkemail`. In case the remote address returns `false`, the validation error is displayed.

Take a look at the screenshot above and you should be able to follow what's happening. And you can, of course, call the URL for validation directly, getting a blank page with either **true** or **false** returned.

Validate, validate, and validate some more!

Make absolutely sure to validate the information provided here. This protects against attackers trying to insert malicious code into your page. We're using `Convert::raw2sql()` and you should too whenever trying to query the database with user-supplied input!

And the final question: is this secure, for example against spammers? Someone can try to brute-force guess the available addresses in our system. But this is at least as much work as simply brute-force sending e-mails to any guessed address. So this shouldn't be a concern.

What would be insecure is adding all e-mail addresses to the page's source and doing some client-side matching based on that. But no one would try to do something like that, right?

Doing more with sessions

If you're doing the validation in the form handler (for example, when checking for duplicate e-mail addresses without JavaScript support), all form data is lost. As a rule of thumb, whenever you need to manually call `Director::redirectBack()`, the form data is lost. We can tackle this with the help of sessions. How do sessions work in general?

By default, HTTP requests are stateless, meaning that there is no history or context. On the one hand this is pretty good performance-wise, as the server doesn't need to keep track of the users and what they are currently doing. On the other hand, this stateless nature is often not sufficient, for example, when users should be able to log in or retain information between requests. For doing that, there is the session concept.

It is generally provided by the programming language used, PHPSESSID when using PHP, but we won't go into the details here. SilverStripe provides a convenient wrapper for sessions:

- `Session::set('SessionValue', true)` sets the value `true` on the key `SessionValue`. You can use as many keys as you need and you can also store strings, arrays, and even (serialized) objects—calling `serialize()` on them.
- `Session::get('SessionValue')` retrieves the value we've just set.
- If you've set multiple keys, you can retrieve all of them with `Session::get_all()`.
- In order to remove values from the session, you can either use `Session::clear('SessionValue')` to clear a single key or `Session::clear_all()` for all of them.

Okay, that's easy, but where can we actually make use of it? Let's go back to our form.

Time for action – using the session in forms

SilverStripe provides an easy fix to restore that lost form input with the help of session.

1. Open the `MembercardPage_Controller` and add the highlighted lines:

```
if($error){
  if($data['Zip'] == 0){
    $data['Zip'] = '';
  }
  if($data['Phone'] == 0){
    $data['Phone'] = '';
  }
  Session::set('FormInfo.Form_Form.data', $data);
  Director::redirectBack();
  return;
}
```

2. That's it—reload the form and try it out.
3. Now that it is working, do the same for the `RentPage_Controller` (but only add the session line, as the `$Data` statements are specific to the member card page).

What just happened?

While processing the form, our method notices that something doesn't validate. So we highlight the error in the form, save the provided information in the session, and redirect back to the form. When it's rendered, the session data is automagically picked up and inserted. Both correct and invalid data are filled in so that corrections can be made easily.

The clever part is using the correct session key so that the magic can work—for example, `FormInfo.Form_Form.data`. Both `FormInfo.Form_` and `data` are fixed, only the part in between is variable. But that's simply the function name where we're setting up the form. So if we had called it `EmailForm()` we'd have `FormInfo.Form_EmailForm.data`.

Existing session data is simply overwritten; you don't need to manually clear it out.

The two `if` statements ensure that empty numeric fields are not automatically changed to zero after redirecting. This happens automatically, so we simply change them back, so the users are not confused.

How to store members in the database

The functionality we want to achieve with our member card is clear. But there are four possibilities to implement this—which one is the right approach in your opinion?

- Creating a new `DataObject`, not connected to the existing `Member` class.
- Subclassing the existing `Member` class, and adding new fields as necessary.
- Adding fields to the existing `Member` class via a `DataObjectDecorator`, while continuing to work with the functionality of the original Member class.
- Replacing the `Member` class with your own implementation. Yes, you can configure that with `Object::useCustomClass("Member", "Membercard");`.

This is actually a pretty tricky decision and there's no definite solution. But here are some arguments for and against the different solutions:

- `Member` already has some of the most important fields (`FirstName`, `Surname`, and `Email`), so we could simply reuse them.
- Member card entries will not use all fields; `Password` for instance would be redundant.

- Would we want to add people to the `Member` class if they actually aren't ever supposed to login or do any other actions planned for this kind of object?
- Subclassing `Member` is generally discouraged—remember that core would use the "old" version. Better add additional fields via a decorator, so that the class is used both in your custom code and in core.
- There are additional modules for working on the `Member` class, but not custom `DataObjects`—for example for sending newsletters.
- The `Member` class can also be managed in the backend under `/admin/security/`. However, there is a pretty similar concept for other objects which we'll cover in the next chapter. Take a look at both possibilities and decide which one will fit your needs best.

Pop quiz – advanced forms

Now that we've worked through some more form concepts, take a moment to think about whether the following statements are true or false:

- You can only save data to the database from a form
- Sessions are needed to preserve inputs when doing client-side validation
- Client- and server-side validation can work together via Ajax calls
- With SilverStripe it's easy to provide a fully-customizable search function

Summary

In this chapter we've taken a better look at forms and how to make the most of them.

We've covered how to add a search box to our page, how to actually search, and how to style the results the way we want them. We've explored how to save form data to the database—a feature we've been missing for quite some time. Additionally, we learned how to make our form validation even more sophisticated and user-friendly while still keeping it simple.

Although there are many more things you can do with forms, you now know the most important features—enabling you to build whatever you want.

The decision on how to implement the member card depends on your requirements. We've used a custom `DataObject` for this example, but in the next chapter we'll change that! We'll also introduce an easy way of accessing the data you saved to the database.

10
Targeting the Whole World

Our project is nearly complete, except for dealing with non-English-speaking visitors. The bar manager has been frequently complaining that visitors can't understand our page. While they like the layout and general structure, they'd also love to understand its content. Luckily SilverStripe supports this requirement well, so we can easily take care of the problem.

In this chapter we'll take a look at how to make our website available to everyone, or at least how it can be done. It's hardly feasible or economically reasonable to provide a translation for every existing language.

Basically there are three major areas to cover:

- Configuring the CMS to support our globalization efforts
- Changes to the code, both in templates as well as Model and Controller
- Making the content available in more than one language and localization

So let's conquer the mysteries of the final chapter...

Globalization in general

Don't worry, we'll keep this short and simple. There are the two main concepts. Both are often summarized under the term globalization:

- **Localization**: A system supports a particular language, currency, metric system, date and time formats, formatting of numbers, and so on—often summarized as a locale; or you might want to say a software has been localized. A locale can be more specific than a language. You might, for example, discern between British and American English.

Currently, we've localized our installation to American English and there is no explicit support for other locales.

- **Internationalization**: A system that supports multiple locales and you can add new ones without touching the underlying source code. Internationalization is done once for a piece of software—you would for example replace hard-coded texts with variables that can be replaced at runtime. Internationalized software can then be localized for different regions—by providing the necessary placeholders.

L10n, i18n, g11n—what the heck?

In case you stumble upon these terms: To keep it short and simple, the rather long notions we've covered so far, are often abbreviated. Take the first and the last character, count the ones in between, and leave them out. Then internationalization becomes i18n, localization L10n (capitalized L so you can't easily mistake it for i18n) and globalization g11n.

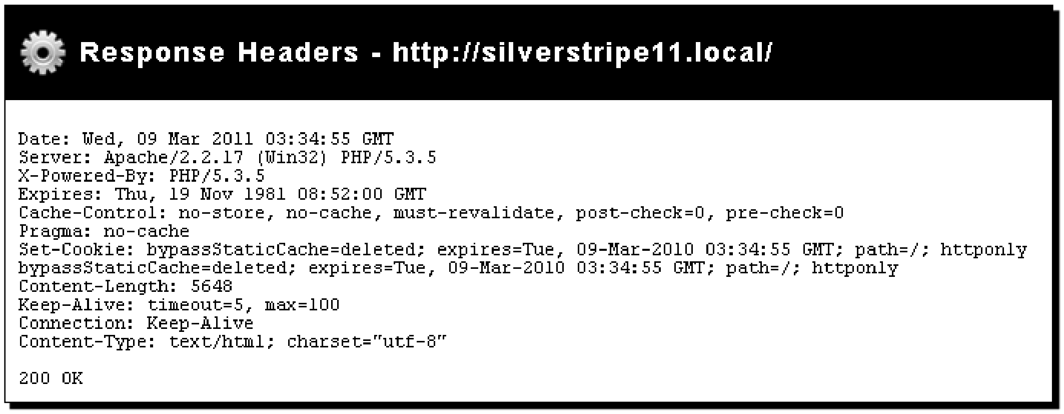
Depending on your locale, the required characters will vary. The available characters are defined by the character set. It defines which sign is represented by which binary code—to limit the required space, there's typically only a subset of all available signs supported. In English this is generally not a concern as more or less all character sets support all required signs. But as soon as you want to use German umlauts, French accents, or Chinese characters, things get a little more complicated.

However, there is an encoding standard, namely Unicode, which should support all meaningful signs in the long term—you're very unlikely to encounter a sign that isn't supported as of today. The Unicode encoding we're using is UTF-8, which is the most common one, and we've been far-sighted enough to use it from chapter two onwards without explicitly knowing it.

The character encoding is important wherever strings are used or exchanged, this includes code and template files, database storage, HTTP transmissions, browsers, and any other clients. To avoid problems, all parts of the system must use the same encoding (or explicitly transform strings into another one). Specifically we've told our database (in the configuration file in `mysite/_config.php`) to use UTF-8 internally: `MySQLDatabase::set_connection_charset('utf8')`, and also our templates in the file `themes/bar/templates/Page.ss`: `<meta charset="utf-8"/>`.

Any file you're using should also be UTF-8 encoded, including templates, PHP, and JavaScript files. Most editors support the setting of a default encoding.

SilverStripe does the rest by automatically adding the right page headers, specifically the HTTP response character encoding providing the "answer" for a client request. The following screenshot was taken with the Web Developer browser plugin, `http://chrispederick.com/work/web-developer/`, available both for Firefox and Google Chrome. Only the second line from the bottom is relevant for the encoding:



```
Date: Wed, 09 Mar 2011 03:34:55 GMT
Server: Apache/2.2.17 (Win32) PHP/5.3.5
X-Powered-By: PHP/5.3.5
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: bypassStaticCache=deleted; expires=Tue, 09-Mar-2010 03:34:55 GMT; path=/; httponly
bypassStaticCache=deleted; expires=Tue, 09-Mar-2010 03:34:55 GMT; path=/; httponly
Content-Length: 5648
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset="utf-8"

200 OK
```

Globalization in the CMS

Now that we're done with the initial preparations, it's time to actually enable globalization.

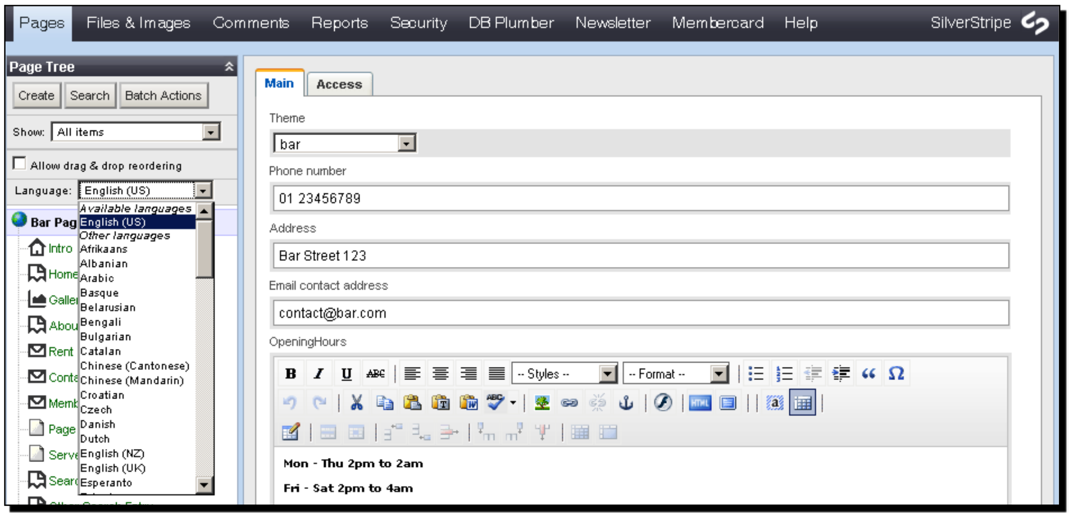
Time for action – configuring the CMS for globalization

We'll start off with the database driven CMS. First we need to internationalize the system by adding some configuration options. We can then add the different localizations in the CMS:

1. Add the following in your configuration file and rebuild the database afterwards:

   ```
   Translatable::set_default_locale('en_US');
   Object::add_extension('SiteTree', 'Translatable');
   Object::add_extension('SiteConfig', 'Translatable');
   ```

2. Now there should be a new drop-down above the site tree where you can select one of the available languages:



Show: All items

Language: English (US)

Theme: bar

Phone number: 01 23456789

Address: Bar Street 123

Email contact address: contact@bar.com

OpeningHours:

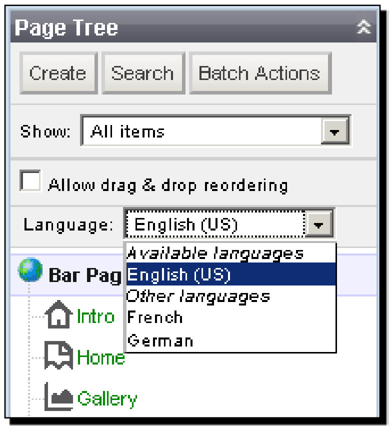
Mon - Thu 2pm to 2am

Fri - Sat 2pm to 4am

3. In case that's a little more than you need and you only require English (US), German (Germany), and French (France), add the following code to your configuration:

```
Translatable::set_allowed_locales(array(
  'en_US',
  'de_DE',
  'fr_FR',
));
```

4. That should limit your options to the ones you actually need:



Show: All items

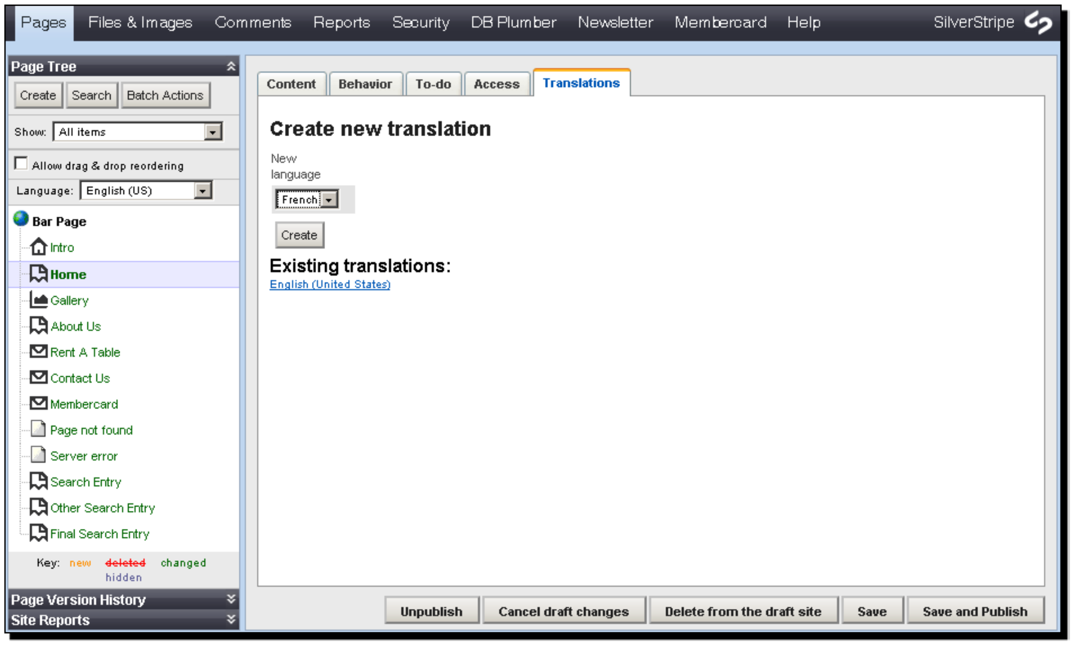
Language: English (US)

5. The template's `$ContentLocale` (which we added in the second chapter to `themes/bar/templates/Page.ss`) is set appropriately.

6. Now we're ready to create content in the CMS in different languages. You can follow two approaches here:

    - Use the drop-down to switch the `SiteTree` to another language. Then you can start creating pages in the chosen language independently from the already existing ones. Translations can follow a totally different structure than the default locale. The CMS's interface looks the same as before.

    - Translate existing pages into another language while sticking to the original structure. On the already existing page go to the **Translations** tab, select the desired language and **Create** the new page. Here you can also see the translation(s) already existing.

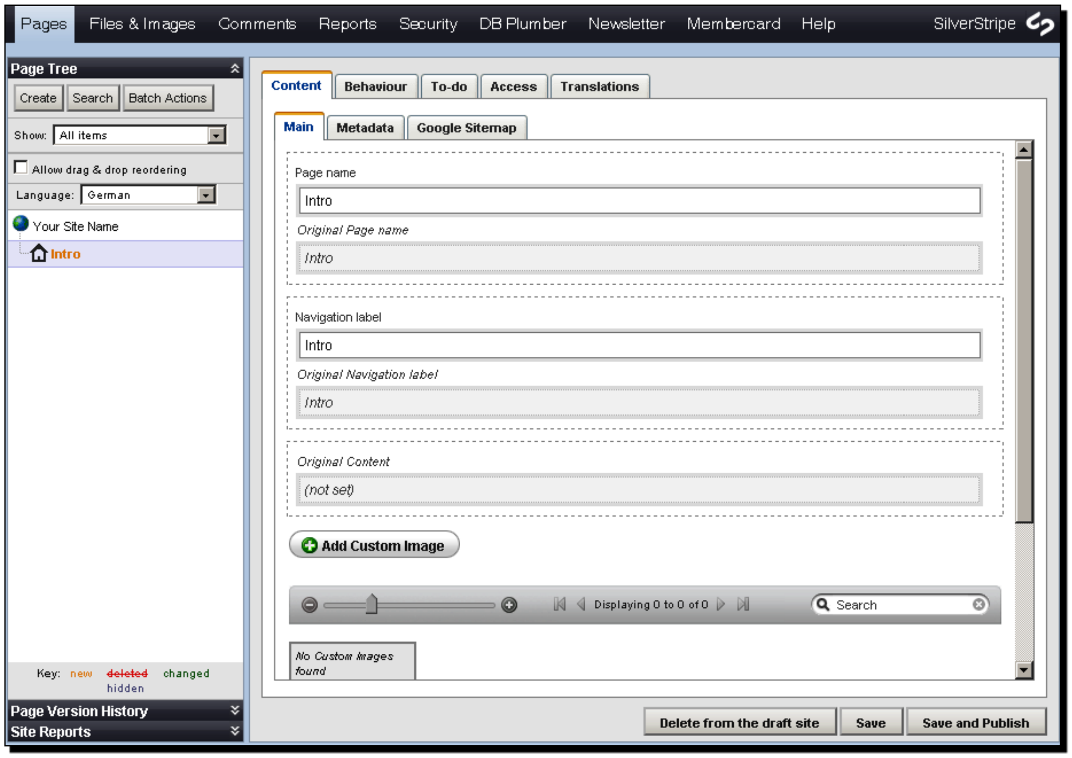


Create new translation

New language: French

Existing translations:
English (United States)

After adding a translation the CMS interface looks quite different. All editable fields are prefilled with the default language's values (including any configuration settings such as **Show in menus?**, and so on) and these values are also displayed as a read-only field below each editable one. This makes it dead simple to keep everything synchronized.



Page name: Intro
Original Page name: Intro
Navigation label: Intro
Original Navigation label: Intro
Original Content: (not set)

7. We're done, except for one little detail: page URLs need to be unique. If the original page's URL was `/intro` the German translation would automagically be `/intro-de-DE`. You should definitely change that, for example to `/einstieg`, as we want our URLs to stay pretty and readable.

8. The only exception is the homepage `/` as it doesn't have an editable URL. Accessing translations can be done by calling `/?locale=de_DE`.

What just happened?

Most of the code is pretty intuitive, but let's go over it quickly:

- `set_default_locale()` sets the language which should be selected by default.
- `add_extension()` enables translations for both pages (a subgroup of `SiteTree`) and the site-wide configuration, or any other class you activate `Translatable` on.
- `Translatable` is actually a regular data object decorator. It adds some fields to a `DataObject` so that the objects can be saved in different languages. Additionally they can also be independently published and versioned. `Translatable` isn't enabled by default to keep database records lean and clean.
- `set_allowed_locales()` limits the available locals, as we've already seen.

To get an overview of all the available languages and their codes, open `sapphire/core/i18n.php` and find the variable `$all_locales`. The general structure is two (or sometimes three) lowercase letters for the language, followed by an underscore and finally the two uppercase letter definition of a locale. Examples include `en_US`, `en_GB`, `en_AU`, and so on.

While URLs default to using dashes (`/intro-de-DE`), everywhere else the locale is underscored (`de_DE`).

Globalization in the code

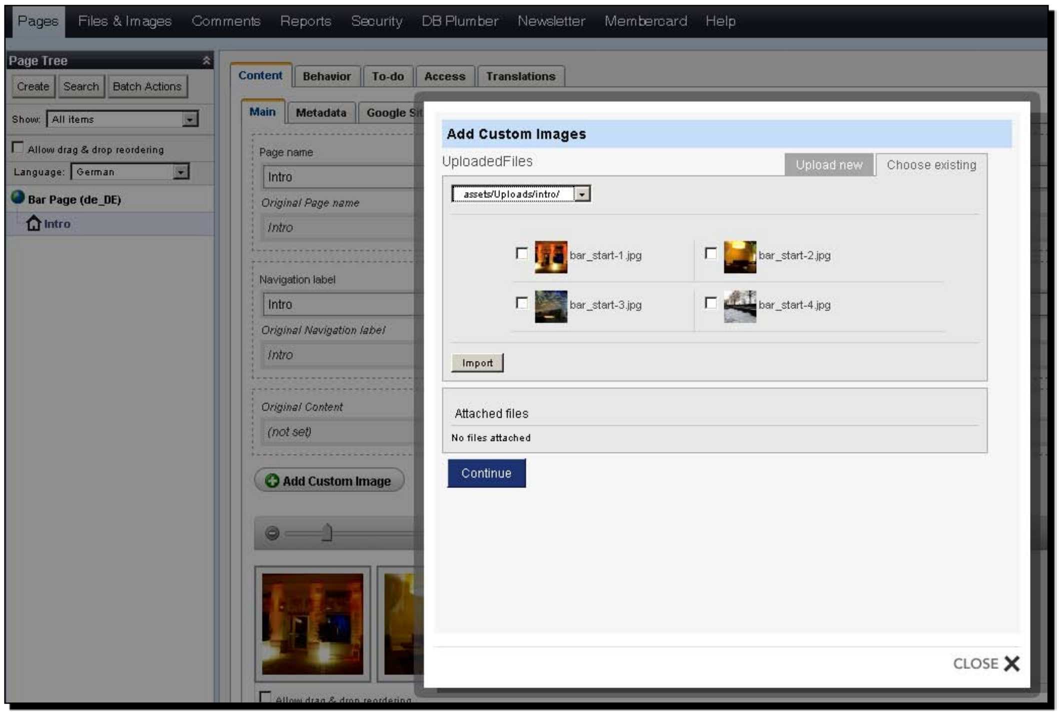
That's it? Not just yet, this is only part of the job. But let's survey the current "problem" through a practical example.

Have a go hero

Create a translation for the `SiteConfig` and the intro page—for example in German (Germany), so that we can see the "missing" pieces in action:

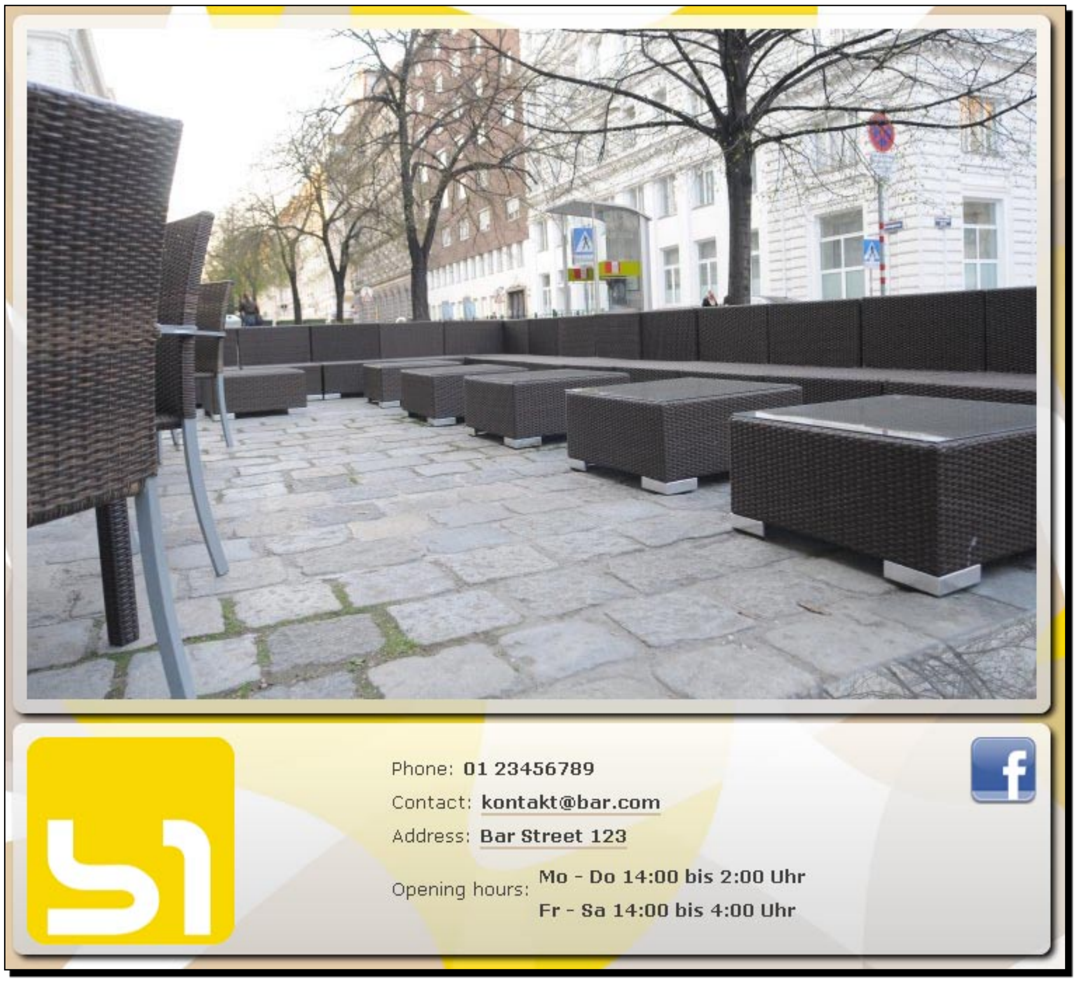
1. Go to the already existing **Intro** page and create a new locale through the **Translations** tab. Set the URL to `/einstieg`.

2. Add the same images as on our English page: Click on **Add Custom Image** and then select them through the **Choose existing** tab.



3. Add the required information to the `SiteConfig` page in the CMS—note that we've named it **Bar Page (de_DE)** to make it crystal clear which locale we are currently in. While the phone number and address will most likely stay the same, the opening hours of our bar will need to be translated, and we could also provide a localized e-mail address.

4. After saving the German pages, go to `/einstieg`, which should now look like this:



Phone: 01 23456789

Contact: kontakt@bar.com

Address: Bar Street 123

Opening hours: Mo - Do 14:00 bis 2:00 Uhr
Fr - Sa 14:00 bis 4:00 Uhr

Now we have a page that's half English and half German—translating the database entries is only part of the job. So let's start to globalize our templates as well and get rid of the hard-coded strings such as **Phone**, **Contact**, and so on.

Localizing and internationalizing the templates

Before we start to code, let's take a short look at how strings can be internationalized. Instead of writing:

```
<li>Phone: <b>$SiteConfig.Phone</b></li>
```

You must change the line to:

```
<li><% _t('PHONE','Phone') %>: <b>$SiteConfig.Phone</b></li>
```

We're calling the function `_t()`—the first parameter is a unique identifier (per template file), while the second one is the default value. So in the previous example, our identifier is `PHONE` and the default text is `Phone`. The function must be wrapped between `<%` and `%>`, the parameters must be quoted, and they may contain spaces. By convention, the identifier is in UPPERCASE and should have a meaningful name.

Starting our globalization efforts

Now that we've covered the basics, let's put this into practice to complete the previously "half-translated" example.

Time for action – globalizing the intro page

Let's get back to our intro page and add the `_t()` function. Additionally we'll need to provide our locales and also add a little locale selector to the `Page_Controller` class.

1. Open up the file `themes/bar/templates/Includes/BasicInfo.ss` and make the following (highlighted) changes:

```
<a href="home">
  <img src="$ThemeDir/images/logo.png" alt="Logo" id="logo"/>
</a>
<ul id="details-first">
  <li>
    <% _t('PHONE','Phone') %>:
    <b>$SiteConfig.Phone</b></li>
  <li>
    <% _t('CONTACT','Contact') %>:
    <a href="contact">
      <span class="mail">$SiteConfig.Email</span></a>
  </li>
  <li>
    <% _t('ADDRESS','Address') %>:
    <a href="location">$SiteConfig.Address</a>
  </li>
</ul>
<div id="details-second">
  <div class="left"><% _t('OPENINGHOURS','Opening hours') %>:</
div>
```

```
  <div class="right">$SiteConfig.OpeningHours</div>
</div>
<a href="http://www.facebook.com/pages/">
  <img src="$ThemeDir/images/facebook.png" alt="Facebook"
    id="facebook"/>
</a>
```

2. Create a file `themes/bar/lang/en_US.php` with the following content:

```
<?php

global $lang;

$lang['en_US']['BasicInfo.ss']['ADDRESS'] = 'Address';
$lang['en_US']['BasicInfo.ss']['CONTACT'] = 'Contact';
$lang['en_US']['BasicInfo.ss']['OPENINGHOURS'] = 'Opening hours';
$lang['en_US']['BasicInfo.ss']['PHONE'] = 'Phone';

?>
```

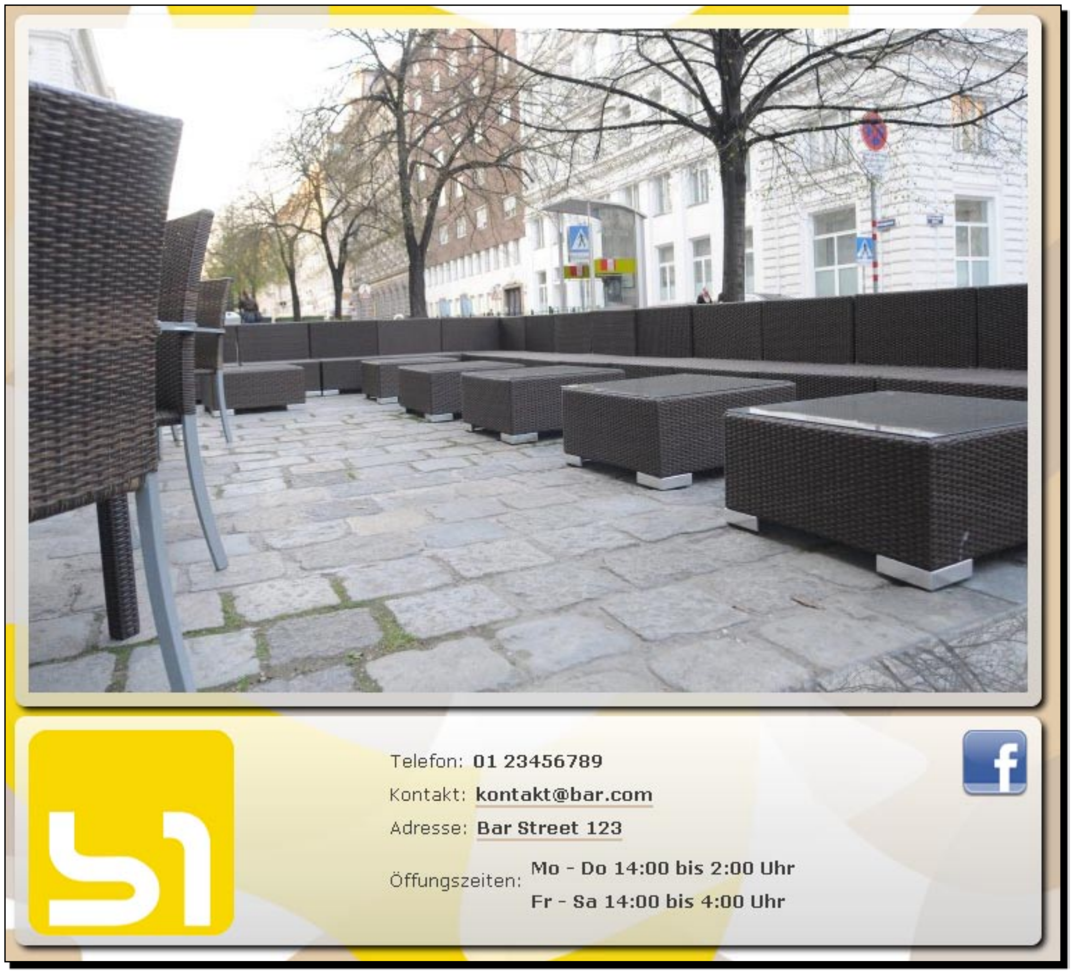
3. And create a file `themes/bar/lang/de_DE.php` with the following content:

```
<?php

global $lang;

$lang['de_DE']['BasicInfo.ss']['ADDRESS'] = 'Adresse';
$lang['de_DE']['BasicInfo.ss']['CONTACT'] = 'Kontakt';
$lang['de_DE']['BasicInfo.ss']['OPENINGHOURS'] = 'Öffungszeiten';
$lang['de_DE']['BasicInfo.ss']['PHONE'] = 'Telefon';

?>
```

4. In the Controller of the file `mysite/code/Page.php`, inside the `init()` function, add the following code:

```
if($this->dataRecord->hasExtension('Translatable')){
  i18n::set_locale($this->dataRecord->Locale);
}
```

5. That's it. Reload the English Intro page, which should look just like in the previous chapters. Then go to our new `/einstieg` page, which should now be completely in German:


Telefon: 01 23456789
Kontakt: kontakt@bar.com
Adresse: Bar Street 123
Öffungszeiten: Mo - Do 14:00 bis 2:00 Uhr
Fr - Sa 14:00 bis 4:00 Uhr

What just happened?

Let's start with the Controller. If the current record has been translated, select the correct locale in all calls to the `_t()` function: `i18n::set_locale($this->dataRecord->Locale);`

The code globalization itself is done by the `i18n` class and not the decorator. That's why you need to set the correct locale in it and not in `Translatable`.

[In case you want to use a single language, there's an even simpler solution: Instead of applying this selector to the Controller, you could replace it with `i18n::set_locale('de_DE');` in `mysite/_config.php`.]

After going through the `init()` method, the template is inspected and the `_t()` method calls are found. Let's assume the identifier `PHONE` is found and the current locale is German (Germany):

- Sapphire will try to locate the file `themes/bar/lang/de_DE.php`. The translation file is always in the same base folder as the `_t()` function, for example `mysite/lang/`, `sapphire/lang/`, and so on. For themes it's in the specific subfolder.
- Inside that file it will look for the variable `$lang['de_DE']['BasicInfo.ss']['PHONE']`.
- `$lang` is a multidimensional array containing all the translations. The first dimension is the locale, the second one the filename (without the path), the third and last one is the identifier. That's also why identifiers only need to be unique inside their respective file.
- If the whole file or the specific variable isn't found, the default text will be returned instead.
- Otherwise the value of the variable is inserted.

For English (US) and any other locale it's working just the same. So it's actually pretty easy to globalize our templates.

[Is it necessary to have the `en_US.php` locale file, isn't that already covered by the fallback text? Strictly speaking it's not required, but it's generally used as the "template" for other translations: Copy the file, rename it and add the translated values. Additionally, instead of editing strings in the template, you could use the locale to override the default values.]

To recap:

1. Internationalize by adding the `_t()` method calls. This should be done right from the start.
2. Localize by providing the specific locales. This can be done whenever the need for a specific locale arises.

Are you stuck with the English version?

If case you forget to set `i18n::set_locale()` in the Controller, the translate method would always return the default locale—English (US). If you're having this problem, check that you set the locale correctly or try to set it statically in the configuration file.

Generating locale files

Hand crafting the locale files is a pretty boring task and it's easy to forget some variables. SilverStripe development should be fun, and that's why there's a tool to generate the default locale for you, already prefilled with the default values—which you can then use as a "template".

All you need to do is call the URL `/dev/tasks/i18nTextCollectorTask` to generate the default locales for the entire site, or `/dev/tasks/i18nTextCollectorTask/?module=templates` for just the `templates/` folder (or whichever directory you want to target). If the `lang/` folder and default locale don't yet exist, they'll be created, otherwise new values are added to the existing file.

There are just two things you should note before calling the tool:

- It's both a time and memory intensive process. Be sure to set your PHP configuration's `memory_limit` to 128 MB and `max_execution_time` to 90 seconds at least. Otherwise the generation of the locale might abort before finishing.
- You'll need PHPUnit, `https://github.com/sebastianbergmann/phpunit/`. It's generally used for testing code, but is required for the text collector task. Assuming you've installed PEAR (if you haven't, take a look at the installation instructions in the appendix), you need to type the following commands into your command line client:

```
pear channel-discover pear.phpunit.de
pear channel-discover components.ez.no
pear channel-discover pear.symfony-project.com
pear install phpunit/PHPUnit
```

This is only required on the machine where you generate the locales—most likely your local development machine but not on your production server.

[This is only an enhancement. If you feel comfortable to create locales manually or don't want to change your configuration, there's no need to.]

Localizing and internationalizing the PHP code

Now that we've covered the templates, let's take a look at the PHP code. Again we're using the `_t()` function, but is has got some more options compared to the template version. An example would be

```
_t('RentPage.OPENING', 'Opening Hour (hour)', PR_MEDIUM, 'CMS text');
```

Let's step through the parameters:

1. The first difference is that the identifier now has a namespace—the part before the dot. By convention this should be the main class name which is also the filename. In the template this isn't necessary as the filename is automatically used instead, but not so in PHP files. Here you should explicitly define the namespace.
2. The second parameter is the default text and just the same as in the template. It's actually optional, but you should definitely provide it.
3. The optional priority defines how important and widely used the translation is. This allows translators to prioritize their efforts. Possible values include `PR_HIGH`, `PR_MEDIUM`, and `PR_LOW`. An empty parameter is between `PR_MEDIUM` and `PR_LOW`.
4. The final parameter is the context. It should give translators a better idea what the string is used for so they can find an appropriate translation.

What should you do if you want to use the same string in two classes, let's say `Here` and `There`? You definitely shouldn't define the translation twice—remember **Don't Repeat Yourself** (**DRY**). If both classes share the same parent class, use this as the namespace. If they don't, select one class and use it as the namespace in both classes.

Time for action – translating the rent form

Let's put this into practice by translating our custom rent form. We could also work on any other page, but this one covers some tricky situations and is therefore a good example.

1. Apply the highlighted changes to the `getCMSFields()` function inside the `mysite/code/RentPage.php file`:

```
public function getCMSFields(){
  $fields = parent::getCMSFields();
  $fields->addFieldToTab(
    'Root.Content.Datepicker',
    new NumericField(
      "OpeningHour",
      _t(
        'RentPage.OPENING',
        'Opening Hour (hour)',
        PR_MEDIUM,
        'CMS text'
      )
    )
  );
  $fields->addFieldToTab(
    'Root.Content.Datepicker',
    new NumericField(
      "ReservationAdvance",
      _t(
        'RentPage.ADVANCE',
        'How many days one can book in advance',
        PR_MEDIUM,
        'CMS text'
      )
    )
  );
  return $fields;
}
```

2. In the `Form()` method we'll need to change the date format in the JavaScript selector. While we're using `YYYY-MM-DD HH:MM` in the English version, we'll switch to `DD.MM.YYYY HH:MM` which is the common form in German. Additionally we'll need to translate some more strings such as the days of the week, months, and so on. Therefore, we'll extend our `#Form_Form_Datetime` field to this:

```
    $("#Form_Form_Datetime").AnyTime_picker({
      format: "' . _t('RentPage.JSDATE', '%Y-%m-%e %H:%i') . '",
  firstDOW: 1,
      dayAbbreviations: [
        "' . _t('Emailer.SUN', 'Sun') . '",
        "' . _t('Emailer.MON', 'Mon') . '",
        "' . _t('Emailer.TUE', 'Tue') . '",
        "' . _t('Emailer.WED', 'Wed') . '",
        "' . _t('Emailer.THU', 'Thu') . '",
        "' . _t('Emailer.FRI', 'Fri') . '",
        "' . _t('Emailer.SAT', 'Sat') . '"
      ],
      monthAbbreviations: [
        "' . _t('Emailer.JAN', 'Jan') . '",
        "' . _t('Emailer.FEB', 'Feb') . '",
        "' . _t('Emailer.MAR', 'Mar') . '",
        "' . _t('Emailer.APR', 'Apr') . '",
        "' . _t('Emailer.MAY', 'May') . '",
        "' . _t('Emailer.JUN', 'Jun') . '",
        "' . _t('Emailer.JUL', 'Jul') . '",
        "' . _t('Emailer.AUG', 'Aug') . '",
        "' . _t('Emailer.SEP', 'Sep') . '",
        "' . _t('Emailer.OCT', 'Oct') . '",
        "' . _t('Emailer.NOV', 'Nov') . '",
        "' . _t('Emailer.DEC', 'Dec') . '"
      ],
      labelTitle: "' . _t('Emailer.DATETIME', 'Select a Date and
    Time') . '",
      labelYear: "' . _t('Emailer.YEAR', 'Year') . '",
      labelMonth: "' . _t('Emailer.MONTH', 'Month') . '",
      labelDayOfMonth: "' . _t('Emailer.DAY', 'Day') . '",
      labelHour: "' . _t('Emailer.HOUR', 'Hour') . '",
      labelMinute: "' . _t('Emailer.MINUTE', 'Minute') . '",
      labelSecond: "' . _t('Emailer.SECOND', 'Second') . '",
  earliest: start,
      latest: end,
    });
```

3. For the actual fields do the following, but we'll only show the first few lines as the rest is just the same:

```
$datetime = new TextField(
  'Datetime',
  sprintf(
    _t(
      'RentPage.DATETIME_LABEL',
      "Date and time (until 12:00 a.m. for the same day, at most
%s days in advance)<noscript><br/>Format: YYYY-MM-DD HH:MM</
noscript>"
    ),
    $this->ReservationAdvance
  ) . SPAN
);
$firstName = new TextField(
  'FirstName', _t('RentPage.FIRSTNAME_LABEL', 'First name') . SPAN
);
```

4. And don't forget the send button:

```
$send = new FormAction('sendemail', _t('Emailer.SEND', 'Send'));
```

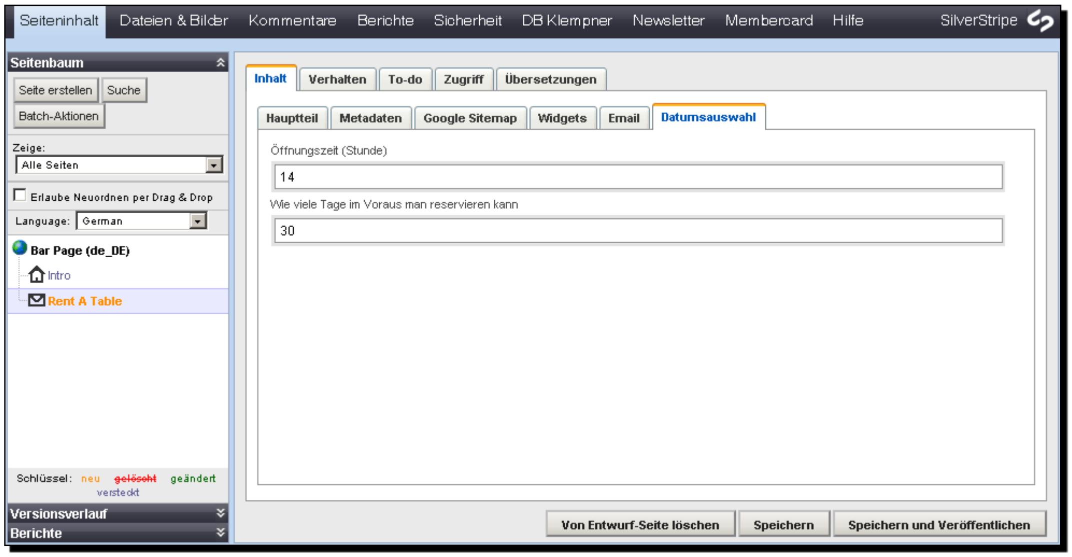
5. In the method `sendemail()` you'll only need to change a single line:

```
$form->addErrorMessage(
  'Count',
  _t(
    'RentPage.COUNT_ERROR',
    'The number of people must be between 1 and 100, please
correct your input'
  ),
  'error'
);
```

6. Call `/dev/tasks/i18nTextCollectorTask/?module=mysite` to generate the default locale or create it by hand.

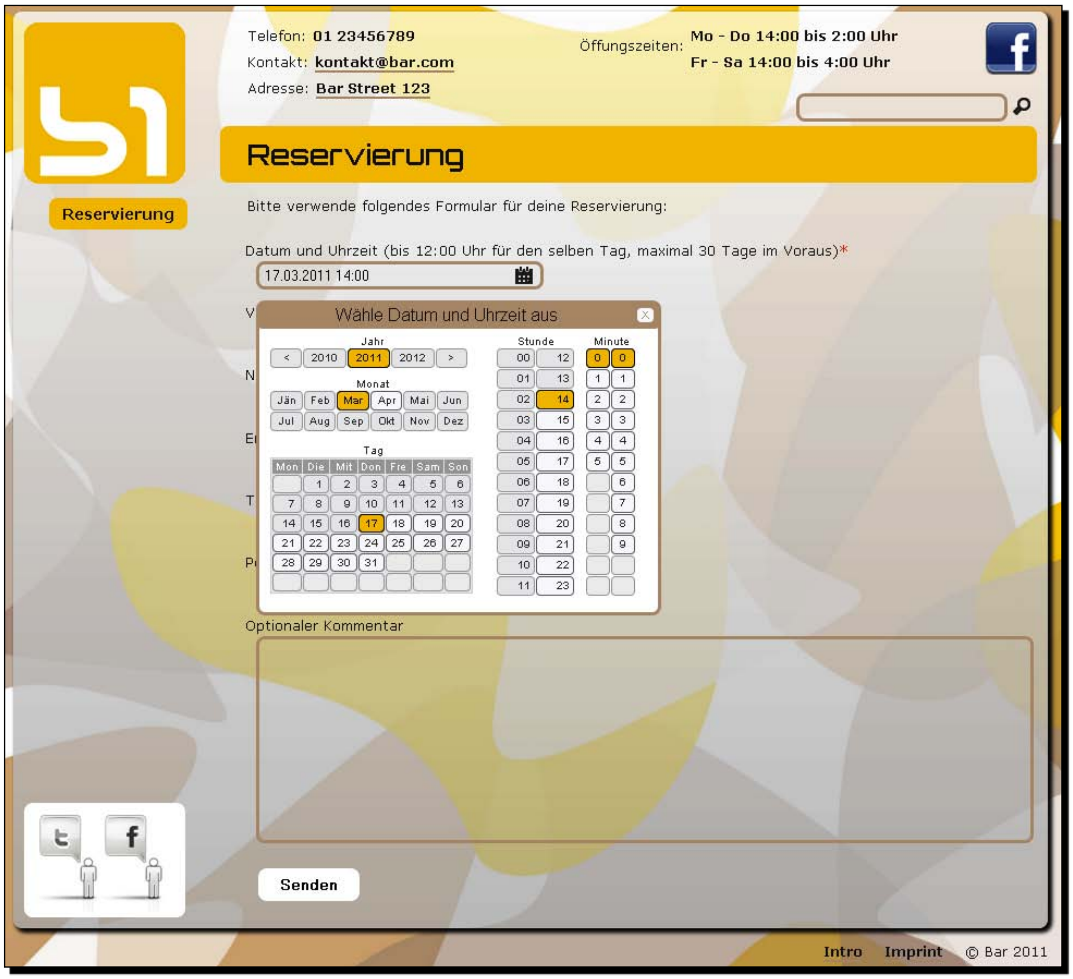
7. Copy the file `mysite/lang/en_US.php` to `mysite/lang/de_DE.php` and translate it.

8. Translate the existing **Rent A Table** page in the CMS and give it a unique URL, for example `/reservierung`. As we've also translated the backend labels, it will look like the following screenshot if we switch the CMS to German. In the CMS, click on the **Profile** link on the very bottom to switch the backend language for the current user (you'll need to reload the CMS to put the setting into action).



Öffnungszeit (Stunde)
14
Wie viele Tage im Voraus man reservieren kann
30

9. Now to the page `/reservierung` which should look like this—fully translated:


Telefon: 01 23456789
Kontakt: kontakt@bar.com
Adresse: Bar Street 123
Öffnungszeiten: Mo - Do 14:00 bis 2:00 Uhr
Fr - Sa 14:00 bis 4:00 Uhr

Reservierung

Bitte verwende folgendes Formular für deine Reservierung:

Datum und Uhrzeit (bis 12:00 Uhr für den selben Tag, maximal 30 Tage im Voraus)*
17.03.2011 14:00

Wähle Datum und Uhrzeit aus

Optionaler Kommentar

What just happened?

Let's go over the noteworthy parts of our example.

Point one is straight forward, we're using the textbook `_t()` function with all of its parameters. From now on we'll only use the shorthand version to save some space, but adding priority and context in general is definitely encouraged.

While the second point is just adding a lot of translations, the third point is getting more interesting.

- Here we have a dynamic part in our string: we're accessing the database value `$this->ReservationAdvance`.
- We could work around this by providing two translations and embedding the dynamic value in between them. However, this doesn't sound like an optimal solution and gets more complicated as the word order in different languages will most likely vary.
- A more elegant and versatile solution is the use of placeholders within the translated string. PHP's built in method for this is `sprintf()`. Its first argument is a string including one or more type specifiers—these always start with a percent sign, for example `%s` to insert a string. The method's second argument replaces the first type specifier, the optional third argument the second specifier, and so on.
- Let's take a look at an example: `sprintf(_t('Class.BETWEEN', 'Hello %s %s!'), $salutation, $name);` If the first variable has the value `Mr.` and the second one `Jon Doe`, the resulting string would be: `Hello Mr. Jon Doe!`

> The same is possible in the template, but only with a single type specifier. It looks like this: `<% sprintf(_t('TITLE',"The page's title is '%s'"),$title) %>`

The rest of the form's code is straightforward again.

Concerning the use of different namespaces: we're assuming that the strings are generally page-specific. However, as we're using the datetime picker on our `Membercard` page as well, we're putting the translations of it into the parent's namespace (`Emailer`). The strings will definitely be reused in our other form.

Let's also take a short look at the generated translation file `mysite/lang/en_US.php`. You'll find some entries which are similar to the template's ones:

```
$lang['en_US']['RentPage']['COMMENT_LABEL'] = 'Optional Comment';
```

However, there are others which don't provide a string, but an array instead:

```
$lang['en_US']['RentPage']['ADVANCE'] = array(
  'How many days one can book in advance',
  PR_MEDIUM,
  'CMS text'
);
```

This is actually the result of the following piece of code that we've added and you can see how the different arguments are mapped:

```
_t(
  'RentPage.ADVANCE',
  'How many days one can book in advance',
  PR_MEDIUM,
  'CMS text'
)
```

There are also two entries per page type—one for the singular and one for the plural. This means that the names of the page types are nicely translated in the CMS:

```
$lang['en_US']['ContactPage']['PLURALNAME'] = array(
  'Contact Pages',
  50,
  'Plural name of the object, used in dropdowns and to generally
identify a collection of this object in the interface'
);
$lang['en_US']['ContactPage']['SINGULARNAME'] = array(
  'Contact Page',
  50,
  'Singular name of the object, used in dropdowns and to generally
identify a single object in the interface'
);
```

Note that the priority constants have the following integer values internally: `PR_HIGH` 100, `PR_MEDIUM` 50, and `PR_LOW` 10.

So that's all the "magic" behind the translation of PHP code in SilverStripe.

Localizing and internationalizing JavaScript

Our "Reservierungs" page is finished except for the validation messages in `mysite/javascript/RentPage.js`. But that's the last step—promise!

JavaScript's `_t()` function is similar to the PHP version. As we've already progressed so far, let's take a quick look at the complex version including dynamic parts (multiple type specifiers are possible):

```
ss.i18n.sprintf(
  ss.i18n._t('VALIDATION.Email', '"%s" isn't a valid email address'),
  email
);
```

Again we're using namespaces—you may want to group your JavaScript code into logical modules sharing the same strings, for example `VALIDATION.email` is a variable containing the (invalid) address provided.

There are some things to bear in mind when doing translations in JavaScript, but we'll cover them while doing our practical example.

Time for action – translating the rent form's JavaScript

First off, the bad news: the translation files currently can't be generated from JavaScript files, we'll need to do that by hand. You'll also need to include some more files, but we'll go through that step by step:

1. First you need to include the i18n library to make the `_t()` function available. Include the following line in your the `init()` method of `mysite/code/Page.php`:

```
Requirements::javascript(SAPPHIRE_DIR . '/javascript/i18n.js');
```

2. Additionally we'll need to specify the directory including the JavaScript translation files. This should be placed in the `init()` method as well. Note that we've set up the constant in the `mysite/_config.php` file (if you're translating a module you'll need to use the correct path):

```
Requirements::add_i18n_javascript(PROJECT_DIR . '/javascript/
lang');
```

3. Open up the file `mysite/javascript/RentPage.js`. We'll only need to change the strings in the `messages:` part of the code. The globalized version should look like this:

```
messages: {
  Datetime: {
    required: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Required'),
      ss.i18n._t('VALIDATION.Datetime_Field')
    )
  },
  FirstName: {
    required: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Required'),
      ss.i18n._t('VALIDATION.Firstname_Field')
    )
  },
  Surname: {
    required: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Required'),
      ss.i18n._t('VALIDATION.Surname_Field')
    )
  },
  Email: {
    required: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Required'),
      ss.i18n._t('VALIDATION.Email_Field')
    ),
    email: ss.i18n._t('VALIDATION.Email')
  },
  Phone: {
    required: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Required'),
      ss.i18n._t('VALIDATION.Phone_Field')
    ),
    digits: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Digits'),
      ss.i18n._t('VALIDATION.Phone_Field')
    ),
    range: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Range_Digits'),
      ss.i18n._t('VALIDATION.Phone_Field'),
      6,
      15
    )
  },
  Count: {
    required: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Required'),
      ss.i18n._t('VALIDATION.People_Field')
    ),
    digits: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Digits'),
      ss.i18n._t('VALIDATION.People_Field')
    ),
    range: ss.i18n.sprintf(
      ss.i18n._t('VALIDATION.Range_Numbers'),
      ss.i18n._t('VALIDATION.People_Field'),
      1,
      100
    )
  }
}
```

4. Create the file `mysite/javascript/lang/en_US.js` with the following content:

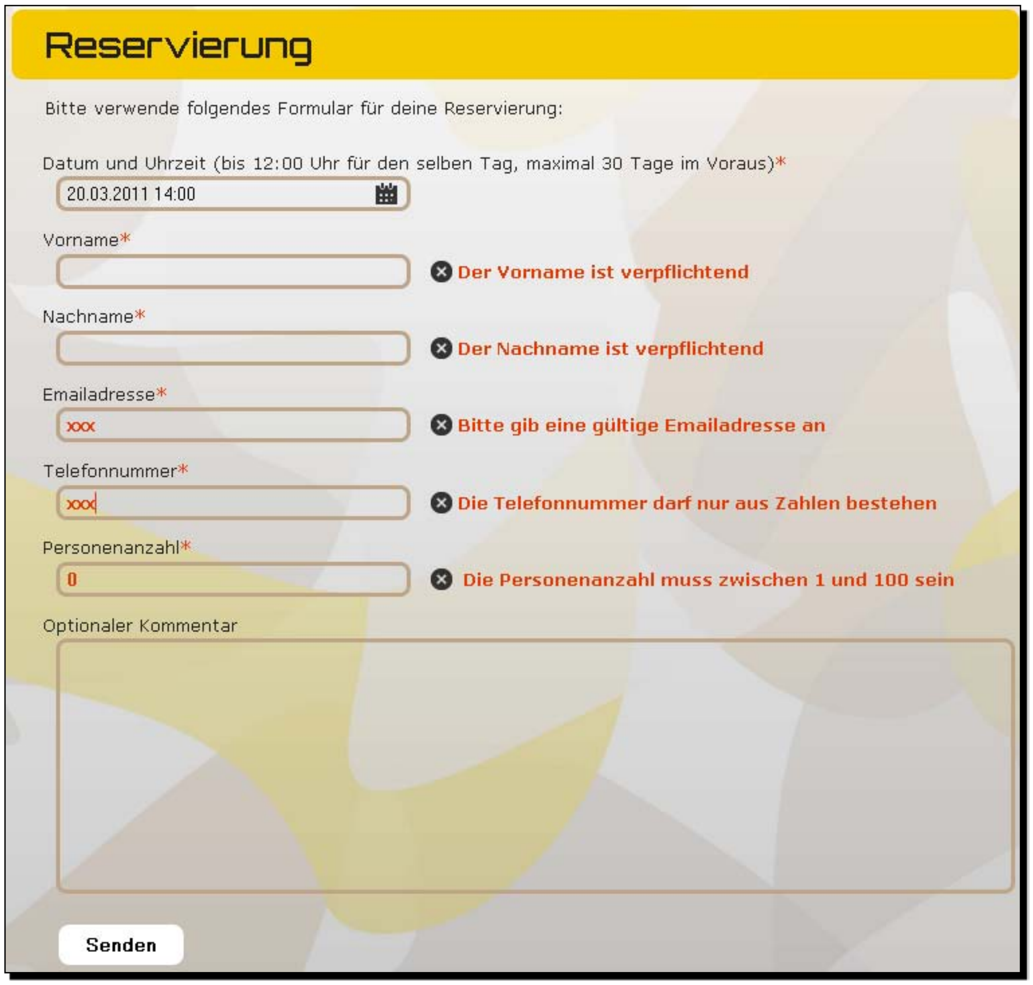
```
if(typeof(ss) == 'undefined' || typeof(ss.i18n) == 'undefined'){
  console.error('Class ss.i18n not defined');
} else {
  ss.i18n.addDictionary('en_US', {
    'VALIDATION.Required': '%s is required',
    'VALIDATION.Digits': '%s must only consist of numbers',
    'VALIDATION.Range_Digits':
      '%s must have between %s and %s digits',
    'VALIDATION.Range_Numbers': '%s must be between %s and %s',
    'VALIDATION.Email': 'Please provide a valid email address',
    'VALIDATION.Datetime_Field': 'The Date and time field',
    'VALIDATION.Firstname_Field': 'The first name',
    'VALIDATION.Surname_Field': 'The surname',
    'VALIDATION.Email_Field': 'The email address',
    'VALIDATION.Phone_Field': 'The phone number',
    'VALIDATION.People_Field': 'The number of people'
  });
}
```

5. Also create a translation of it in `mysite/javascript/lang/de_DE.js`: (the `en_US` in the file is not an error but a "trick", we'll come to that when discussing the code):

```
ss.i18n.addDictionary('en_US', {
  'VALIDATION.Required': '%s ist verpflichtend',
  'VALIDATION.Digits': '%s darf nur aus Zahlen bestehen',
  'VALIDATION.Range_Digits':
    ' %s muss zwischen %s und %s Stellen haben',
  'VALIDATION.Range_Numbers': ' %s muss zwischen %s und %s sein',
  'VALIDATION.Email': 'Bitte gib eine gültige Emailadresse an',
  'VALIDATION.Datetime_Field': 'Das Datums und Uhrzeit Feld',
  'VALIDATION.Firstname_Field': 'Der Vorname',
  'VALIDATION.Surname_Field': 'Der Nachname',
  'VALIDATION.Email_Field': 'Die Emailadresse',
  'VALIDATION.Phone_Field': 'Die Telefonnummer',
  'VALIDATION.People_Field': 'Die Personenanzahl'
});
```

When using three `%s` in a string, note the space at the start. When using more than one type specifier, the first one is sometimes not picked up correctly and the extra space corrects the problem.

6. Reload the page—our JavaScript error messages should look like this (only showing the relevant part of the page):


Reservierung

Bitte verwende folgendes Formular für deine Reservierung:

Datum und Uhrzeit (bis 12:00 Uhr für den selben Tag, maximal 30 Tage im Voraus)*

20.03.2011 14:00

Vorname*

Der Vorname ist verpflichtend

Nachname*

Der Nachname ist verpflichtend

Emailadresse*

xxx

Bitte gib eine gültige Emailadresse an

Telefonnummer*

xxx

Die Telefonnummer darf nur aus Zahlen bestehen

Personenanzahl*

0

Die Personenanzahl muss zwischen 1 und 100 sein

Optionaler Kommentar

What just happened?

First off we're loading the i18n JavaScript implementation so that we can use it on the client-side. Additionally we need to specify where the specific translation files are located, by convention `javascript/lang/` inside the module's folder is used.

Each translation has its own file and two of them are sent to the client: first the default translation, which is English (US), as a fallback; and then the currently active locale.

Our translation schema in this example is quite elaborate. To keep repetitions to a minimum, we're dynamically combining quite a lot of strings. For example, we're only defining "is required" once and dynamically inserting the specific field name. When defining a range we're inserting three dynamic values: first the field name, then the lower bound, and finally the upper bound. When using this for a single form, it may be overkill but the more forms your site has, the more it will pay off.

[In this example we've left out the default strings. First to save some space, but also to minimize repetitions even further. Defining "%s is required" five times (in this form alone) doesn't sound too appealing after all. Especially if you might want to change it in the future.]

In the default locale we're first checking if SilverStripe's i18n features have been correctly loaded—if not we're adding an error message to the console. The actual translation strings are organized in dictionaries, which you're adding to the current page as follows:

```
ss.i18n.addDictionary('en_US', {
```

[Be very careful where you place commas and colons. JavaScript is very picky about this.]

Now to our little "trick". How does the client figure out which of the two locales should be used? Before HTML5 you would have inserted the following code into the `init()` method (where we've loaded i18n and defined the directory of the JavaScript locales):

```
Requirements::insertHeadTags('
  <meta http-equiv="content-language" content="' .
      i18n::get_locale() . '"/>
');
```

However this is deprecated in HTML5 and should instead be added to the `<html>` tag instead, which we did back in chapter two: `<html lang="$ContentLocale">`.

Unfortunately SilverStripe's i18n implementation currently doesn't recognize this. To keep our markup nice and clean, we're employing a little trick, which also demonstrates how i18n is actually applied:

- As we've said, the default locale is loaded first, followed by the page-specific one.
- So the `mysite/javascript/lang/en_US.js` file loads all required values which are then overwritten by `mysite/javascript/lang/de_DE.js`.
- If the translation for a given string is missing, it will be left as it is.

This concludes the translation of strings. We're now able to fully translate every aspect of SilverStripe.

Have a go hero

If the language you want to use is incomplete or you spot an error, you can easily fix it. SilverStripe provides a central translation server (free registration required), where you can update translations or fetch the latest version: `http://translate.silverstripe.org`. While languages such as German or Spanish are already production ready, others need some more work. Specifically they're waiting for YOUR contribution!

Getting the right content

Now that we've translated our pages, we need to make sure our visitors also get the right content. We'll take a look at a usability issue we've ignored so far.

Time for action – switching the locale

At the moment, English-speaking visitors can find the home page of our site directly under `/`. In contrast our German-speaking visitors would need to know the URL `/einstieg` to be taken to the German intro page. Let's provide a link so visitors can easily switch the language.

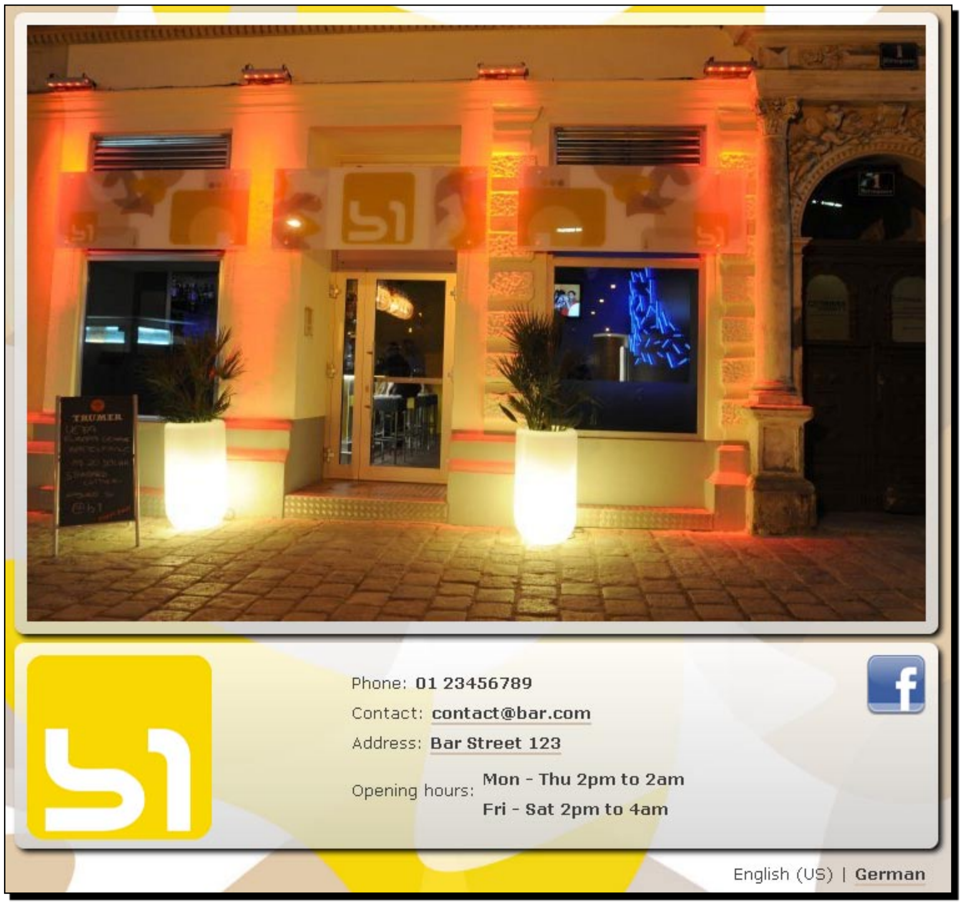
1. At the bottom of `themes/bar/templates/Layout/IntroPage.ss` add the following code:

```
<div id="intro-page">
  <% include LanguageSwitcher %>
</div>
```

2. Create the file `themes/bar/templates/Includes/LanguageSwitcher.ss`:

```
<aside id="locales">
  $Locale.Nice
  <% if Translations %>
    <% control Translations %>
      |
      <a href="$Link" hreflang="$Locale.RFC1766" title="$Title">
        $Locale.Nice
      </a>
    <% end_control %>
  <% end_if %>
</aside>
```

3. With some additional styling the English Intro page looks like the following screenshot. If a user clicks on the **German** link, he'll be redirected to the German intro page.



Phone: 01 23456789

Contact: contact@bar.com

Address: Bar Street 123

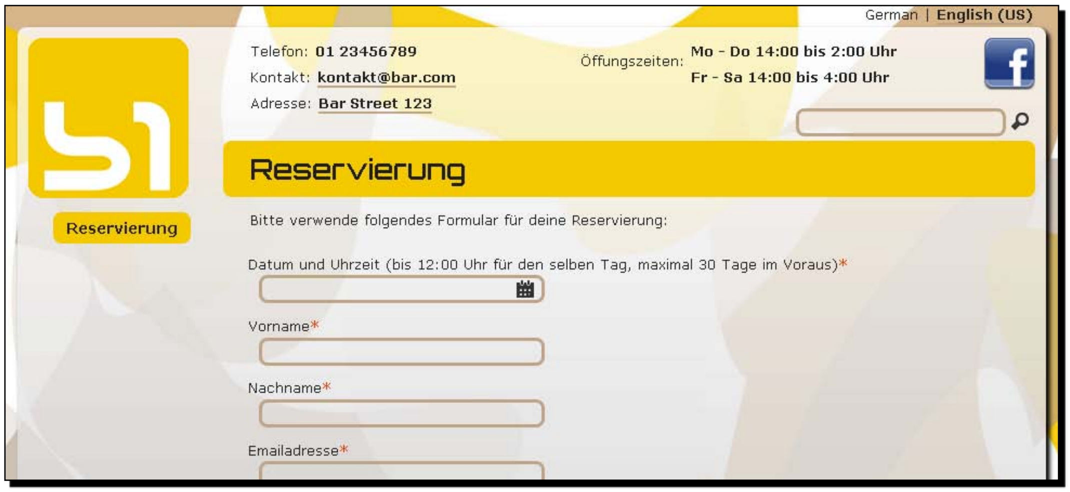
Opening hours: Mon - Thu 2pm to 2am
Fri - Sat 2pm to 4am

English (US) | German

4. Add the following code to `themes/bar/templates/Layout/Page.ss` and `themes/bar/templates/Layout/GalleryPage.ss`—heres' the relevant section with the changes highlighted:

```
<div>
  <img src="$ThemeDir/images/background.jpg" alt="Background"
id="background"/>
</div>
<div id="content-page">
  <% include LanguageSwitcher %>
</div>
<section id="content" class="transparent rounded shadow">
```

5. Reload one of the non-intro pages which should look like the following screenshot (this time in German):



German | English (US)

Telefon: 01 23456789
Kontakt: kontakt@bar.com
Adresse: Bar Street 123

Öffungszeiten: Mo - Do 14:00 bis 2:00 Uhr
Fr - Sa 14:00 bis 4:00 Uhr

Reservierung

Bitte verwende folgendes Formular für deine Reservierung:

Datum und Uhrzeit (bis 12:00 Uhr für den selben Tag, maximal 30 Tage im Voraus)*

Vorname*

Nachname*

Emailadresse*

What just happened?

There are just some new template elements so let's go through them:

- `$Locale` returns the code of the currently active locale, `en_US` for example.
- `$Locale.Nice` returns the full name of the locale, "English US" for example.
- The `Translations` control element contains all other locales available for the current page. You can use it to loop over the other translations of the current SiteTree element—in our case only German.
- Inside the control structure you can access all of its elements, including the title (`$Title`), URL (`$Link`), locale (`$Locale`), and so on.
- `$Locale.RFC1766` returns the locale's code with a dash instead of an underscore, en-US for example. You need this form to provide a valid `hreflang` attribute, showing user agents that the link leads to another locale (even though it's currently mostly ignored).

There's just one point to note: if you're on a page and there is no translation for it (yet), only the current locale is displayed. No link is provided as there is no page to link to.

Pop quiz – if or when should you globalize

Now that we know how to globalize our code, if or when should you do it?

1. Avoid it if possible, only do it as late in the development process as possible—it's a lot of work after all.
2. You should always globalize right from the start of a project.
3. It depends, but you should rather err on the side of globalizing too much and too early.
4. It depends, but you should rather err on the side of globalizing too little and too late.

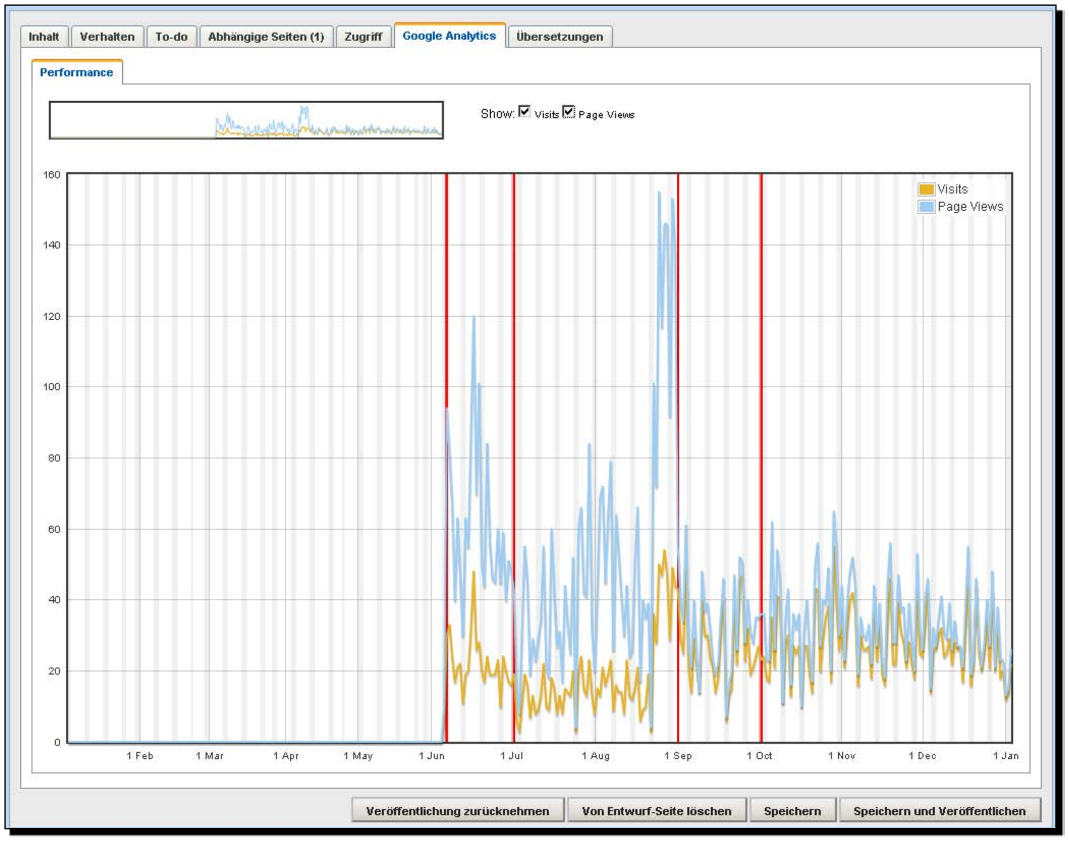
Where to go from here

By now, we've covered all the major points to get up and running. But of course there's more. Here are some hints as to what you could do after finishing the book to progress even further.

More modules

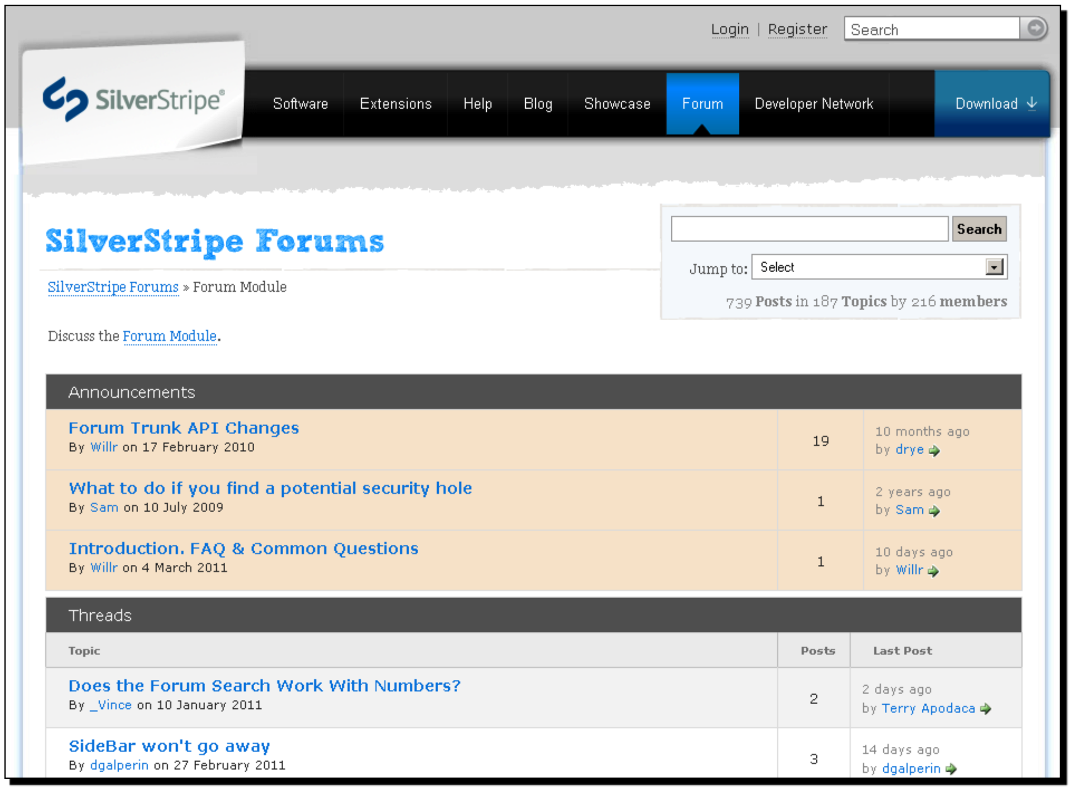
There are many modules we've not been able to cover. Depending on your specific use case, your needs will vary greatly. For a complete list see `http://silverstripe.org/modules/`. But here's a list of a few very popular ones:

- Google Analytics (`http://silverstripe.org/google-analytics-module/`): If you're using Google Analytics, like we did in our example, you can integrate a high level overview of visitors and page views directly in the CMS. It even allows you to zoom in and out of the data. The module looks like the following screenshot—showing off the German version as we're in the globalization chapter:



- Blog (`http://silverstripe.org/blog-module/`): If you want to integrate a blog into your website, that's the module to use. It provides a summary page with previews and the full length articles as well as a tag cloud, archive and RSS widget. Use it to show the visitors of our bar what's going on!

- Forum (`http://silverstripe.org/forum-module/`): Adds a forum to your page. It may be a good idea to provide a more interactive communication channel like this, so visitors can share their stories, ideas, criticism, news, and anything else relevant for our page. The forum module also powers SilverStripe's official forum: `http://silverstripe.org/forums/`—which integrates very well into the rest of the site, as you can see in this example:



SilverStripe Forums

SilverStripe Forums » Forum Module

739 Posts in 187 Topics by 216 members

Discuss the Forum Module.

Announcements		
Forum Trunk API Changes By Willr on 17 February 2010	19	10 months ago by drye
What to do if you find a potential security hole By Sam on 10 July 2009	1	2 years ago by Sam
Introduction. FAQ & Common Questions By Willr on 4 March 2011	1	10 days ago by Willr

Threads

Topic	Posts	Last Post
Does the Forum Search Work With Numbers? By _Vince on 10 January 2011	2	2 days ago by Terry Apodaca
SideBar won't go away By dgalperin on 27 February 2011	3	14 days ago by dgalperin

- E-Commerce (`http://silverstripe.org/ecommerce-module/`): If our bar is doing really well, you may want to sell branded goods—for example T-shirts, glasses, and so on. With the help of SilverStripe's e-commerce module this can be easily done and it tightly integrates into the rest of the system. It allows you to add a product catalogue, shopping cart, and supports different payment providers. So there's nothing holding us back from pursuing this opportunity as well.

Adding more ideas and features to your page

Modules cover a wide array of features and tasks, but there's even more to make your site feature-rich, appealing and stable:

- SilverStripe supports both unit and integration tests, to ensure that your code is working as expected and to ensure it stays that way. Take a look at `http://doc.silverstripe.org/sapphire/en/topics/testing/why-test` for getting started.
- Does your system need to interact with other systems, for example to exchange some new information? Both SOAP and REST (the two major standards in this area) are well supported—see `http://doc.silverstripe.org/sapphire/en/reference/restfulservice`.
- Do you want to broadcast your latest updates? SilverStripe makes it super easy to provide an RSS feed. Either take a look at the `RSSFeed` class or the official documentation at `http://doc.silverstripe.org/sapphire/en/reference/rssfeed`.
- Do you need scheduled tasks, for example sending birthday wishes to your users every night? That's also well supported; just take a look at `DailyTask` or the other `Task` classes.

Regardless of what you want to do, be sure to visit `http://doc.silverstripe.org` for the latest high-level documentation or `http://api.silverstripe.org` for the finer details of the underlying code whenever you're stuck. Now that you know the basic concepts, all you need is the documentation of specific details to conjure up your magic.

If you encounter an error and it's not in your own code, please report it to `http://open.silverstripe.org`. Your help is greatly appreciated and assists to continuously enhance SilverStripe for you and everyone else.

Summary

In this chapter we've taken on globalization, specifically how to implement it in SilverStripe.

Among other things, we've covered what localization and internationalization are, and how to work with different languages in both the CMS and on a code level, including templates, PHP and JavaScript files. We've then discussed how to make globalized content accessible to our users. We also contemplated what to do after finishing the book and how to progress even further.

That's it—go forth and build great websites with SilverStripe!

Pop Quiz Answers

Chapter 2, Customizing the Layout

Pop quiz—placeholders

Adding `$Content` or `$Form` to a page is easy. But are we already able to build a dynamic menu, including all the pages we've created in the CMS?

1. Yes, we can do that using `$URLSegment`.
2. Yes, that's what `$LinkingMode` and `$Link` are for.
3. Yes, `$Breadcrumbs` will do that for us.
4. No, so far we're only able to build static HTML menus.

Answer: Although we've covered the placeholders for building dynamic menus already, we can't build one as we don't have the tools (yet) to loop over the pages. So far our placeholders only fetch the content from the page we're currently viewing. For the moment we're stuck with plain old HTML.

Pop quiz—say hello to the current user

Take a look at the following template snippet:

```
<% if $CurrentMember %>
  Hello $CurrentMember.FirstName $CurrentMember.Surname!
<% end_if %>
```

Assume that you are currently logged into the CMS and have set both the first and surname for the current user. What will this output?

1. The placeholder `$CurrentMember` doesn't exist, so this won't output anything.
2. It will output your name, for example "Hello Philipp Krenn".
3. This will cause an error because of the first line.

Answer: If your answer was option 1, go back to the user part of placeholders.

If you selected option 2, take a look at the structure of if statements. It's a pretty common problem to erroneously add `$` inside a control structure and easily overlooked.

And if you went with option 3, congratulations, you definitely know your way around SilverStripe's template engine.

Chapter 3, Getting "Control" of Your System

Pop quiz—duplication or not?

To illustrate that, let's take a look at the following example:

```
Requirements::css(THEMES_DIR . '/' . $theme . '/css/layout.css');
Requirements::css(THEMES_DIR . '/' . $theme . '/css/layout.css');
Requirements::customCss('.double { font-weight: bold; }');
Requirements::customCss('.double { font-weight: bold; }');
Requirements::customCss('.once { font-weight: bold; }', 'once');
Requirements::customCss('.once { font-weight: bold; }', 'once');
Requirements::customCss('.which { font-weight: bold; }', 'which');
Requirements::customCss('.which { font-weight: normal; }', 'which');
```

Assume that you've added it to the `init()` method, which of the following statements is true?

1. The file `layout.css` is included twice. As there isn't an (optional) unique identifier argument for `Requirements.css()` there isn't anything we can do about it.
2. The CSS class `.double` is added twice in the pager's header section as it doesn't have a unique identifier.
3. The CSS class `.once` is added once in the pager's header section, as the unique identifier avoids the repetition.
4. The CSS class `.which` has a bold font weight as the second entry is ignored.

Answer: Points 2 and 3 are true, while the other two are not:

The file `layout.css` is only added once as SilverStripe is clever enough only to include a specific file a single time. That's also the reason why there isn't an optional ID argument—it's simply not necessary.

In case you're using a unique ID in custom CSS, the last entry is used. You could probably use this feature to cleverly overwrite styles, but don't confuse yourself.

Pop quiz—basic principles

What are the three most important principles of SilverStripe and what does each of them achieve?

Answer: They are

- Model View Controller (MVC)
- Don't Repeat Yourself (DRY)
- Convention over Configuration

Chapter 5, Customizing Your Installation

Pop quiz—environment types

Which of the following statements are true for the three modes you can put SilverStripe into:

1. After the installation, your page is automatically put into development mode so that you can easily debug problems.
2. Development sites should be in development mode, live sites should be in live mode with logging enabled.
3. You can use `Debug::show()` and `Debug::message()` in all three modes to directly output some information into your browser.
4. You need to be logged in as an administrator to call `/dev/build` and `?flush=all` on non-development sites.

Answer: Option one is false. By default no environment type is added to your configuration file and the default is live mode. This is a security conscious decision so that you don't accidentally reveal any sensitive information. However, it can also be a source of frustration for beginners as it makes debugging problems harder.

The second question is true while the third one is false. In live mode no debugging messages are shown. Again this is done for your own protection—forgotten debug statements won't have an impact on your productive pages.

The last point is true again —you don't want any visitor to update your database schema, don't you?

Chapter 6, Adding Some Spice with Widgets and Short Codes

Pop quiz—too many options?

So far we've discussed four methods for extending the CMS:

- Adding custom fields to a page.
- Adding custom fields to the site configuration.
- Creating a widget or using an existing one.
- Creating your own short code.

Take a look at the following requirements. The four approaches have pretty distinct use cases and you'll make your life needlessly hard if you try to achieve something with the wrong approach. Which one(s) fit the requirements best?

1. You want to add the same advertisement on all of your pages for a special offer.
2. You want to display the offer from the previous point, but only on some selected pages.
3. You want to display your special offer right in your `$Content`.
4. You want to display different offers on various pages. Some pages shouldn't contain any advertisement.

Answers: For the first question you should definitely consider adding custom fields to the site configuration. The keywords here are **the same** and **on all of your pages**. You'll only want to create the advertisement once and enable or disable it for your whole site.

Number two could be easily solved with a widget or short code. You'd probably define the advertisement once in the site configuration, which you could then either drag-and-drop onto a page with a widget (fixed position in the sidebar). Or you could place it freely within your `$Content` placeholder through a custom shot code.

Requirement three can only be satisfied with short codes—the keyword here is **right** in your `$Content`.

The last option could be easily solved with custom fields or a widget. You could either fill in the custom, page specific fields (or leave them blank) or activate a widget on the desired pages into which you'd fill in the specific advertisement as a custom field.

Chapter 7, Advancing Further with Inheritance and Modules

Pop quiz—SiteTree, DataObject, or DataObjectDecorator

Selecting the right data-type is crucial for getting the right mix of features and not making your life unnecessarily hard.

Assume that you want to create your own image gallery. You want to easily add images in the CMS. In the frontend thumbnails should be displayed. If one is clicked a bigger version should pop up. Which of our three base classes would you use for achieving this task?

1. Just `SiteTree`: We want to create a new image gallery page, so that's it.
2. Just `DataObject`: All we want to do is show some images so nothing else is needed.
3. `SiteTree` and `DataObject`: We need to extend a new image gallery page (`SiteTree`) with images (`DataObject`)..
4. `SiteTree` and `DataObjectDecorator`: We need a page, so `SiteTree` is necessary. Additionally we want to inject some additional information into the base image class through the `DataObjectDecorator`.

Answers: A DataObject is required for holding images, but it must be connected to a `SiteTree` element to display the pictures on a given page—leaving us with the correct answer number three. Neither SiteTree nor `DataObject` can achieve this on their own. A `DataObjectDecorator` is not required as we can extend the existing image class and don't need to inject information into it.

Chapter 8, Introducing Forms

Pop quiz—true or false

Which of the following statements are true, and which are false?

1. You can select any method name to set up the form, but `Form()` is common.
2. You should always add JavaScript in the Controller to guarantee security—otherwise it can be changed on the client-side.

3. You can only trust server-side validation.
4. You should remove any JavaScript not required on a page.

Answers: Option one is true, but watch out: If you finally return the newly created form with `return new Form($this, 'Form', $fields, $actions, $validator)`, the second argument must match the method's name. This is a very common error.

While option two is false (client-side code can always be manipulated by the user, hence the name), number three is true.

Finally, the last option is obviously true. For bonus points, in which method did we remove the unnecessary JavaScript files? Inside the `init()` function of `mysite/code/Page.php`.

Chapter 9, Taking Forms a Step Further

Pop quiz—advanced forms

Now that we've worked through some more form concepts, take a moment to think about whether the following statements are true or false:

1. You can only save data to the database coming from a form.
2. Sessions are needed to preserve inputs when doing client-side validation.
3. Client- and server-side validation can work together via Ajax calls.
4. With SilverStripe it's easy to provide a fully customizable search function.

Answers: Option one is false. While you'll often use forms when saving data to the database, they are definitely not required. You can load an already existing object or create a new one, and can then set any properties on it. Finally you can store it, by calling the `write()` method on the changed object—whether it's based on a form or not.

The second option is also wrong. In this case sessions are server-side. When doing client-side validation, the input values are never sent to the server so the session can hardly store them. Sessions are only necessary when doing server-side validation due to the statelessness of HTTP.

The last two statements are true—as we've already seen in our example project.

Chapter 10, Targeting the Whole World

Pop quiz—if / when should you globalize

Now that we know how to globalize our code, if or when should you do it?

1. Avoid it if possible, only do it as late in the development process as possible—it's a lot of work after all.
2. You should always globalize right from the start of a project.
3. It depends, but you should rather err on the side of globalizing too much and too early.
4. It depends, but you should rather err on the side of globalizing too little and too late.

Answers: While the answer isn't clear cut, the best approach is somewhere between options 2 and 3. As rule of thumb:

- Modules should always be globalized right from the start. Due to their portability there's a good chance that either you or someone else (in case you've open sourced it) will need a different language than the one you've used. After all modules are there for maximum portability and the locale shouldn't be an exception.
- For site specific code it depends. If you're absolutely sure your site will never be required to support a different locale, you could skip its globalization. However, if there's only a slight chance of ever needing a second locale, you should take care of it right from the start. Making these changes for an already finished site is both tiresome and error prone.

Index