INSTANT

Short | Fast | Focused

Lift Web Applications How-to

Get to know the Lift Web Framework quickly and efficiently using practical, hands-on recipes

Torsten Uhlmann

[PACKT] PUBLISHING

Instant Lift Web Applications How-to

Get to know the Lift Web Framework quickly and efficiently using practical, hands-on recipes

Torsten Uhlmann

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Instant Lift Web Applications How-to

First published: January 2013

Production Reference: 1180113

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

ISBN 978-1-84951-978-6

www.packtpub.com

Credits

Author

Torsten Uhlmann

Reviewers

Richard Dallaway

Marius Danciu

Diego Medina

Acquisition Editor

Jonathan Titmus

Commissioning Editor

Meeta Rajani

Technical Editor

Devdutt Kulkarni

Project Coordinator

Abhishek Kori

Proofreader

Jonathan Todd

Production Coordinator

Prachali Bhiwandkar

Cover Work

Prachali Bhiwandkar

Cover Image

Conidon Miranda

About the Author

Torsten Uhlmann is a German-based freelance Software Craftsman, a husband, and a dad, no dog. He has worked on numerous medium to large software projects over the course of nearly two decades. He has gained insight into many different technologies, from Cobol to Ruby, from Oracle to MongoDB, from programming CICS terminals to developing scalable web applications using a wide range of different technologies.

A few years back he fell in love with Scala as a very expressive language that challenged many of the things he thought he knew about software design. He joined the growing number of Lift committers contributing a port of a showcase application to Java in an effort to open up the framework for multiple programming languages. To this day he greatly enjoys writing performant and scalable Lift applications for his clients, one of them being the secure private network `sgrouples.com`.

Torsten's home on the Web is `http://www.agynamix.de`.

I'd like to thank my wife Silvia for her patience and strong support during the long hours when this book was created. While I sat down having fun writing it, she took care of the real life around us. Thank you for being the great companion and friend God has given to me.

A magnificent thank-you goes to David Pollak, Richard Dallaway, Diego Medina, and Marius Danciu for taking time reviewing the book and making sure what I write is true.

The entire Lift mailing list also deserves a huge thank-you—this is an awesome place to ask questions and get help!

Mark Weinstein, CEO of Sgrouples, thank you so much for allowing me to write this book while we were super busy building our gorgeous application!

And last but certainly not least, I would like to thank the team at Packt Publishing. It was a pleasure working with my reviewers, Meeta Rajani and Priya Sharma. Thank you for the awesome experience!

About the Reviewers

Richard Dallaway is a partner at Underscore Consulting, the UK's leading Scala consultancy, where he specializes in delivering client projects using Scala and Lift. His background is in machine learning applied in the finance, manufacturing, retail, and publishing industries. He is a Lift committer, focusing on the module system, and writes for The Lift Cookbook.

Marius Danciu has been a full-time programmer for the last 10 years. He discovered Scala in 2007/2008 and also learned a great deal of functional programming through Scala. Coming from the world of imperative languages (C/C++, Java), he found functional programming an epiphany. Since then, Marius joined the Lift team working on core parts of the Lift framework. This has been an outstanding experience and motivated him to learn more Scala, functional programming, and more mathematics. However, at his job he doesn't do a lot of Scala coding but works on growing the Scala adoption. Still, he's doing interesting stuff in the area of distributed computing and MapReduce, functional DSL language design, and so on.

Marius is also a co-author on the book *The Definitive Guide to Lift: A Scala-based Web Framework, Apress*.

Diego Medina lives on the mountains of North Carolina with his wife, 2-year old daughter, and their cat. He has been a developer for the past 11 years, and his focus has been on web development, and more specifically, web security.

He is a proud Lift committer and a very active member of the Lift community, answering questions on the mailing list, as well as writing articles on his personal blog.

He currently holds the position of developer in the R&D department at Elemica Inc., where they are using Lift and Scala as main technologies for the next generation of their platform.

I would like to thank Torsten Uhlmann for the opportunity to review such a great book; he has done a great job.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

If you prepare to write a web application these days, you face a plethora of options. You have to decide for a programming language and then select a web framework for it. No easy choice.

In this book we'd like to introduce you to the Lift framework, a full stack web application framework for the Scala language.

At its core, Lift addresses security and usability as much as developer flexibility. It makes it tremendously easy for you to create high-performing, security-enabled, and highly interactive applications. This book helps you through the initial Lift learning curve, to make you more productive at a faster rate.

What this book covers

Preparing your development environment (Simple), guides you through the process of installing all necessary software components and describes their basic behavior. At the end of this recipe you will have a fully working Lift application running on your machine.

Preparing your Eclipse environment (Simple), helps you install all the components you need to develop and run a Lift application. We will guide you through installation, setup, and initial use of the Eclipse development environment together with the Scala IDE plugins for Eclipse.

Saying hello to Lift Boot (Simple), leads you through an initial set of the several Lift application configuration steps you need to master, in order to create a working application.

Designer friendly templates (Simple), introduces you to Lift's way of cleanly separating the HTML view from server-side logic.

Using Lift snippets (Simple), helps you understand the server-side counterpart of designer friendly templates. Snippets are pieces of Scala code that seamlessly plug into the templates and provide dynamic functionality.

CSS selector bindings (Simple), provides an easy and convenient way for Lift snippets to inject server-side logic and data into templates.

Binding dynamic content (Medium), touches, maybe, the most important task in today's web applications, transforming and displaying data from different sources to your users.

Managing page access (Simple), which is one of Lift's security features, is a convenient way to integrate page access control into a menu structure. It gives you central control over the pages served to users depending on the user's status or maybe their status in your application.

Building a dynamic menu structure (Advanced), introduces you to Lift's unique way of extracting URL parameters in a type-safe way so that you can send users to URLs such as `/photos/123/show`.

Lift's MegaProtoUser (Medium), is a customizable user management implementation complete with login form and verification e-mail processing. We will learn how to use and extend its capabilities.

Handling forms (Simple), teaches you how to query the user for data and how to process that data within your application.

Form validation (Simple), guides you through the process of validating user data and presenting error messages.

Using Ajax (Simple), helps you get up to speed with Lift's Ajax integration quickly. You will also learn how you can very easily Ajax-enable any form in your application.

Going real time with Comet (Advanced), introduces you to Lift's Comet support. While Ajax sends user data to the server without a page refresh, Comet push sends server data to the browser. Using Comet enables you to create highly interactive applications that will attract your users.

Lift and MongoDB (Advanced), helps you hop on the NoSQL train with MongoDB. Lift comes with a seamless integration for this particular database—using it is easy and straightforward.

MongoDB and Rogue (Advanced), builds upon the previous recipe and teaches you how to make use of Foursquare's Rogue library for an even easier integration of Mongo into your application. Rogue provides a way to let you create type-safe, easy-to-understand database queries for Mongo.

Building a REST API (Medium), shows you how easy Lift makes it for you to provide clean and secure REST access that is usable from browsers or mobile applications alike.

Integrating Twitter Bootstrap (Medium), teaches you to build your applications using the successful Bootstrap CSS framework along with a sample application ready for you to use.

What you need for this book

To work with the examples in this book you need a Java JDK Version 6 or later installed on your computer. The examples should run on any recent version of Windows, Mac, or Linux. For some of the recipes you need the Mongo NoSQL database installed on your PC or network.

Who this book is for

If you would like to start developing web applications with the Lift framework or are interested in learning more about it, this book is for you. In addition, this book will be a guide for managers, helping them to decide whether the Lift technology is applicable.

We expect the reader to be a little familiar with the Scala programming language. However, we do not assume any existing Lift knowledge.

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: "Now, open `build.sbt` in the same folder and uncomment the line `// scanDirectories := Nil`."

A block of code is set as follows:

```
object MenuGroups {
  val SettingsGroup = LocGroup("settings")
  val TopBarGroup = LocGroup("topbar")
}
```

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

```
{
  val liftVersion = "2.4"
  libraryDependencies ++= Seq(
    "net.liftweb" %% "lift-mongodb-record" % liftVersion,
    "net.liftmodules" %% "mongoauth" % (liftVersion+"-0.3"),
    "ch.qos.logback" % "logback-classic" % "1.0.0",
    "org.scalatest" %% "scalatest" % "1.6.1" % "test",
    "org.eclipse.jetty" % "jetty-webapp" % "7.6.0.v20120127" %
                          "container"
  )
}
```

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

```
--launcher.XXMaxPermSize
256m
-vmargs
-Xms256m
```

```
-Xmx1024m
-XX:PermSize=64m
-Xss1M
-server
-XX:+DoEscapeAnalysis
```

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: "In `MacOS` you need to right-click on the **Eclipse** application and choose **Show Package Content**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

Instant Lift Web Applications How-to

Welcome to Instant *Lift Web Applications How-to*. This book will give you a quick, step-by-step introduction into the world of Lift. It will guide you through the different steps of setting up a Lift application, developing pages using content from a database, and making them really spiffy using Ajax and Comet. We expect that you already know the basics of the Scala programming language (`http://www.scala-lang.org`), but we promise to take it easy and explain new constructs as we go along.

When to use Lift

Lift (`http://www.liftweb.net`) is a full stack web application framework. What that means is that Lift comes with all the tools, utilities, and help to build full-scale web applications, ranging from serving simple web pages to building large applications with lots of Ajax and dynamic data in it. The flipside of this coin is that Lift works in a different way compared to the majority of existing frameworks you may have come across. So before your application development starts, you should make a conscious decision whether Lift is an appropriate tool for that job.

We will discuss some of Lift's awesome core strength in the hope that this knowledge will help you in your decision.

Ok, suppose there is your exciting next web project that you develop for yourself or in a team, and you are on a quest of finding the right tool for the job. Let's look at some of Lift's core strength to help you find an answer.

Lift advertises seven things (`http://seventhings.liftweb.net/`) it sees as its core strength. There's more, but let's look at some of these items first:

- **Security**: Web applications are exposed to the world and have to deal with an ever increasing number of threads your application will be exposed to. It's critical to keep access to your site and to your user's data as secure as you can. Lift brilliantly helps you in that regard, for instance by binding backend functionality to random names in the browser. That way an attacker cannot predict which function to call or which Ajax call to spoof. Lift also properly escapes data sent back and forth between browser and server, protecting you from the **cross-site scripting** (**XSS**) attacks, so injecting malicious data into your database queries becomes very hard. There's much more, in terms of security, that Lift has to offer, for instance things that you would need to develop yourself in other web frameworks. And trust me, security features take a long time to develop properly.
- **Comet and Ajax**: Lift provides superb built-in support for super easy use of Ajax. Comet (Ajax long polling) is a push technique that allows the server to send information to the client browser. The integrated Comet support is a tremendous help when you want to develop a real-time application, the classic example being a chat application. But every site that has the following dynamic parts can benefit from Comet:
  - A shopping site being updated with the real-time availability of items
  - A news ticker broadcasting to connected browsers
- **Lazy loading and parallel rendering**: Lift's architecture provides you with tools to load parts of your data in the background, and when the computation is done, this data is pushed (yes, through Comet) to the browser. Parallel rendering will farm off the processing of annotated parts of your page to parallel, processes and the data will be pushed as soon as a part gets ready.
- **Designer friendly templates**: Lift's page templates are pure XHTML or HTML5; there's no code in them, and they have nothing that an HTML parser wouldn't understand. That has several benefits. For the developer, it's a very clean separation of layout and code where template files contain the markup and Scala classes (known as Snippets in Lift land) contain the code. For the designer, it's the joy of working with a clean template without having a fear of messing up the included code.
- **URL whitelisting**: There's a concept called "SiteMap" in Lift. A **SiteMap** is a list of paths on your site that any client may access accompanied by security restrictions. It's easy to say that the home page may be accessed by any client, but other pages can only be accessed by the logged-in users and some others only by admins. Lift will check this access for you, so there's no chance you forget to integrate that in some of your pages (I've heard sometimes developers are in a rush to meet a deadline, and this is when things like this happen).

- **Representational State Transfer (REST)**: Lift has super easy REST support. REST is an agreed-upon standard by which different applications can communicate. For instance, if your web application needs to support mobile clients, a REST API is one very widely used way to support that. Using Lift you are very well equipped to serve your clients through a REST API.
- **Lift is stateful**: Lift distinguishes itself from other web frameworks by keeping the state of the user's conversation in the server. Of course you could also develop your application stateless, yet this feature makes it much easier to develop interactive applications that do things based on the logged-in user, for example showing this user's photos or posts.

Preparing your development environment (Simple)

So here you are. Eager to get started with your new project, but you just feel overwhelmed by the amount of new things that seem to pile up in front of you.

It might be a daunting task to start developing your first Scala or Lift application. Several pieces need to be clubbed together in the right order to ensure a smooth and functioning environment. In this task we will walk through the different tools step by step. After just a few pages you will have a functioning development environment and will already see the fruits of your hard work in the form of a real and running application.

Getting ready

We expect that you have Java 6 or its newer version installed on your machine. It doesn't matter if you work on Windows, Mac, or Linux; all are fine development environments and very much suited for Lift programming. In this recipe we will show you how to install each software component.

How to do it...

To prepare your development environment perform the following steps:

1. Although it's not strictly needed for the toolchain that we describe, it's still recommended that you should download a standalone version of the Scala programming language. The examples in this book will use version 2.9.1. So go to `http://www.scala-lang.org/`, and download and unpack this archive to a directory of your choice.

2. For our own development we choose `/lang/` as the folder that accumulates these packages. If you don't have permission to create this folder on the root level, you might as well place it under your user's directory at `~/lang/` on Unix or `C:\Users\<username>\lang\` on Windows. Be sure to add `/lang/scala-2.9.1/bin` (substitute with the path you choose) to your `PATH` variable on Mac or Linux, or `C:\lang\scala-2.9.1\bin` to the `PATH` environment variable on Windows. That's all; the Scala language is now installed.
3. To test it, open a new terminal window and type in `scala`. If the `PATH` entry is correct, you should see the Scala **Read-Evaluate-Print-Loop** (**REPL**) come up, which is a great way to test out language constructs interactively.

```
Welcome to Scala version 2.9.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_35).
Type in expressions to have them evaluated.
Type :help for more information.

scala> "Scala is fun".map(_.toUpper).split(" ").reverse.mkString(" ")
res0: String = FUN IS SCALA

scala> :q
tuhlmann@Saratoga:~$
```

The preceding screenshot shows a terminal window running the Scala REPL. You can type in Scala code and get it evaluated right away. Here we took the string `"Scala is fun"`, made it all uppercase, split the string into a list of strings, reversed that list, and made it a string again. All in one line.

4. Now, find yourself a convenient place on your computer where you want to store our Lift project; the `develop/` folder inside your user directory sounds like a good place. Go into that directory or create it, and type in the following command:

   `git://github.com/tuhlmann/packt-lift-howto.git`

   This will download the source code for this book. Now navigate to the folder `/packt-lift-howto//lift_howto_9786_sql_tpl`. Type in the following command from within that folder if you are on Unix:

   `./sbt`

   On Windows, type in the following command:

   `sbt.bat`

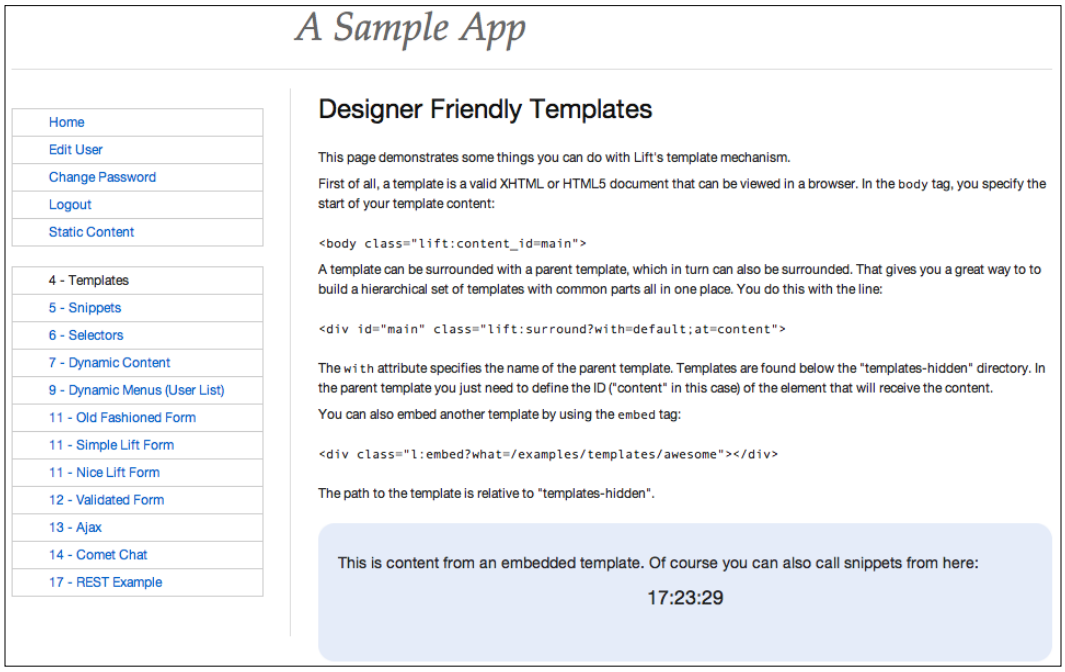
   This Lift template project contains everything to get a Lift project compiled and running. Well, it does not really contain the libraries you need, which you will see when the actual downloading starts. **SBT** (**Simple Build Tool**, available at `http://www.scala-sbt.org/`) reads the `build.sbt` file to know the configuration of your project. It will then check if all the libraries mentioned there and any transitive dependencies are stored in a cache directory (`.ivy2` in your user directory). If not, it will fetch them for you.

5. After a while you should see the SBT prompt (>) indicating you can proceed with further commands. Type in the following command now:

```
container:start
```

This command will compile the sources of this project and will start up a Jetty server at port 8080 so you can see the fruit of your efforts. This template project uses the SQL database "H2" as its backend storage. Since it's Java, you don't have to install any database in advance.

So this template project already shows you a featureful Lift application. It contains user management, user validation via validation e-mail, and, for instance, a "Forgot Password" feature. It protects some content to be visible only to logged-in users and stores all registered users in the database.



6. To stop the Jetty container, enter the following command:

```
container:stop
```

Whew, that was a lot. But we're nearly done. Promise!

Now, let's look at how we make use of JRebel in Lift development.

One constant pain during the development cycle is that you change the source code, it gets compiled, and then it has to be redeployed to the servlet container. Doing that costs time, you usually lose your session information, and it's generally painful. A great tool that can help here is JRebel, which will try to reload any changes you made to your code into the virtual machine. It doesn't always work, but still can prove very helpful. JRebel is a commercial product, but at the time of this writing, you can get a free license for Scala development. Just go to `http://sales.zeroturnaround.com/` and apply for a Scala Developer's license. In the meantime you can download the 30-day trial to use it immediately. For this book's sources I used JRebel 4.6.1.

To install and use it just download the JRebel archive and unpack it (yes, `/lang/` is a good place to put it into). You need to copy the license file you receive into the same folder as the archive. Then go into the Lift template directory and edit the `sbtr` file, which is already configured for JRebel, and set the `JREBEL_HOME` variable to the place you installed it to. Now, open `build.sbt` in the same folder and uncomment the line `// scanDirectories := Nil`. You're done. Now don't use `./sbt` to start the SBT shell but use `./sbtr` to get JRebel goodness.

There's more...

The following list presents some of the SBT commands that you will use a lot. There are more and every plugin adds its own commands, but you usually need to remember only a few, which you need to use repetitively.

Commands	Description
`clean` and `clean-files`	`clean` deletes compiled artifacts, while `clean-files` deletes all downloaded artifacts from the project.
`compile`	This compiles the project.
`test`	This compiles and runs tests.
`container:start`	This starts the Jetty container. If you are using JRebel, this command is enough to get files, which Eclipse compiles, reloaded into the JVM.
`~; compile; container:start`	If you use JRebel but not Eclipse, you can use this command to compile on demand and let JRebel reload the changes.
`container:stop`	This stops the Jetty container.
`~; container:start; container:reload /`	If you do not use JRebel, use this command to make the Jetty container reload on your changes.
`package`	This packs your projects into a deployable WAR file.

It's a wise choice to read a bit about the Simple Build Tool usage at `http://www.scala-sbt.org/`. SBT is simple with respect to its configuration, yet it's very flexible and can do many more things than what we saw here.

Preparing your Eclipse environment (Simple)

Integrated development environments (**IDEs**) provide a plethora of useful features for developers. They speed up the development process and help you understand your code better. One of the leading IDEs is Eclipse (`http://www.eclipse.org`); it's the basis of the official Scala IDE (`http://www.scala-ide.org`).

You can choose from a wide range of editors and IDEs. Different people have different preferences and opinions. The three major IDEs, Eclipse, IntelliJ IDEA, and Netbeans, all come with Scala support. For this book we will choose Eclipse. We use it successfully every day and can recommend using it. But feel free to try out any other editor that you like.

Scala or Lift does not enforce any particular environment, yet we found it helpful to choose one that offers deep support for the language.

Getting ready

This task builds directly on top of the previous task that we explained in the *Preparing your development environment (Simple)* recipe. To avoid confusion and frustration, please make sure to complete the steps given in the previous task (`http://scala-ide.org/`).

How to do it...

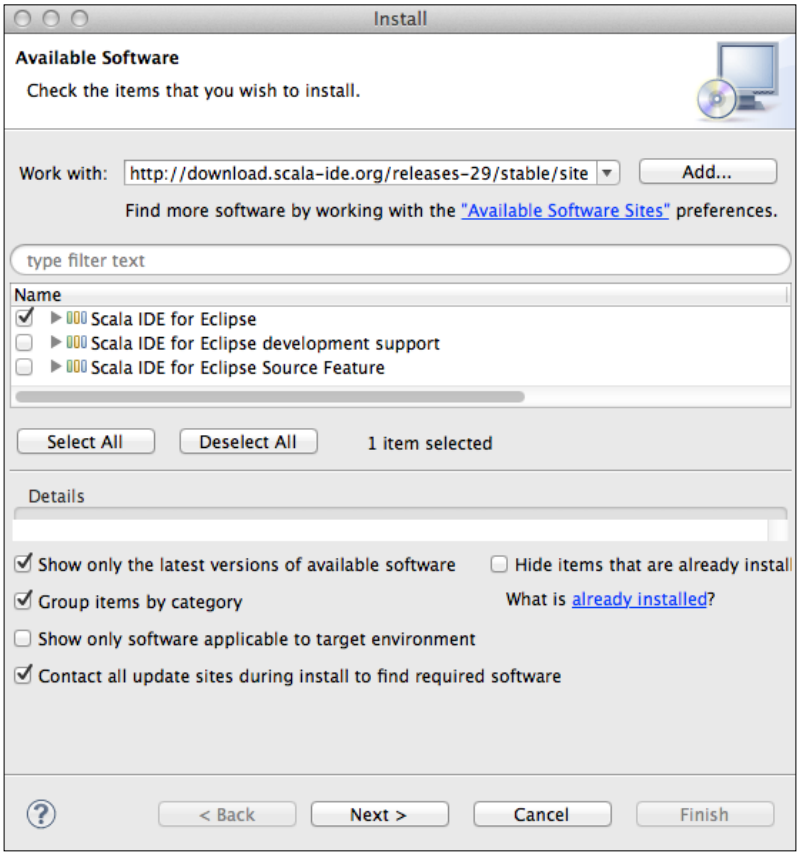
The template project comes bundled with `sbteclipse`, an SBT plugin that will generate your Eclipse configuration. Please change into the template project's folder and perform the following steps:

1. Open an SBT shell by typing in `./sbt`, or `sbt.bat` if you are on Windows, and enter the following command after the prompt comes up:

   ```
   eclipse with-source=true
   ```

   This will generate the Eclipse project structure files. It will also download the source archives for any libraries that your project depends on and links them into the project. If you don't want that, just omit `with-source=true` from the preceding command. Now depending on your Internet connection that might take a while. You will see it's finished when the SBT command prompt appears. Your project is now ready to be imported into Eclipse.

2. To do that let's install a fresh Eclipse installation. Go to `http://www.eclipse.org` and download the latest Eclipse 3.7.2 installation appropriate for your platform. We would like to download the Eclipse Classic installation and add a few other components that we find useful.
3. To install Eclipse, just unpack it into a directory of your choice, for instance `/ewu/`. It is a good idea to rename the `eclipse` folder to something like `Eclipse_Lift`. That distinguishes it from other Eclipse installations you might want to have in the future. But for the sake of simplicity, we just assume you did not rename it.
4. Within the `eclipse` folder you will find an `eclipse` executable file. Just run it. Now after Eclipse starts up, go to **Help** | **Install New Software...**. The following screenshot shows the packages you should install:



The **Scala IDE for Eclipse** plugin is needed in order to do Scala development with Eclipse. Just go to that site and copy the update URL you want to use into the **Eclipse New Software** dialog box. You should start with a stable version of the Scala IDE, and when you feel more confident using it, feel free to switch to the more experimental one.

After installation please restart Eclipse. When it reopens, it will complain that it has too little memory to work properly. We will take care of that in a minute.

There's more...

Aptana (`http://aptana.com/`) is a collection of tools that we highly recommend for any JavaScript or HTML work. It's not strictly necessary for Lift development, but it makes a lot of things easier. As with the preceding Scala IDE, just choose the update URL for Aptana 3 plugins (`http://download.aptana.com/studio3/plugin/install`) and copy that into the **Eclipse New Software** dialog box, which you have to open again. Again the same dance restarting Eclipse. Make sure everything runs fine, then quit Eclipse, and let's bump up its memory footprint.

The process is described in detail at `http://scala-ide.org/docs/user/advancedsetup.html` in the *Eclipse Configuration* section of the Advanced Setup Guide for Scala IDE. Make sure Eclipse is not currently running, then open its `eclipse.ini` file, which contains the Java settings for the JVM that Eclipse runs in. The `eclipse.ini` file can be found in the `eclipse` folder or at `eclipse/Eclipse.app/Contents/MacOS/`. In `MacOS` you need to right-click on the **Eclipse** application and choose **Show Package Content**. On either systems it's a good idea to make a backup copy of that file.

Add or replace the following lines in that file:

```
--launcher.XXMaxPermSize
256m
-vmargs
-Xms256m
-Xmx1024m
-XX:PermSize=64m
-Xss1M
-server
-XX:+DoEscapeAnalysis
```

The values here are suggestions and can be increased further, depending on whether you use a 32-bit or 64-bit system.

Now start Eclipse again and see if no errors occur. If it doesn't start, there's a bug in `eclipse.ini`. It's really fortunate that you made a backup copy, right?

Downloading the example code

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

If all goes well, you can now import the Lift project into Eclipse. To do that perform the following steps:

1. Right-click on the **Package Manager** or **Navigator** view on the left-hand side and choose **Import...**.
2. In the next dialog select **General** | **Existing Projects into Workspace** and click on **Next**.
3. Click on the **Browse** button next to **Select the project's root directory** and find the root directory of the template project (`lift_25_sbt11_sql_tpl`), and click on **Open**.
4. In the **Import** dialog box you should now see your chosen project ready to be imported. Click on **Finish**.

In Eclipse click on **Window** | **Open Perspective** and choose the Scala perspective. The left-hand side shows the package explorer with your project loaded and hopefully no compile errors. Eclipse does compile your files on save and will show you any compilation errors in the bottom view. But even before you compile, it will analyze your code and give you helpful tools, especially when you don't know the source code or the libraries you're working with.

Take some time and play around with the freshly set up environment. Look at the different menus, look at the source code of the template application, try to change it, and see if Eclipse can compile it.

Saying hello to Lift Boot (Simple)

If you have been developing applications, and in particular web applications, for a while, you probably have come across long XML configuration files. In more traditional web application frameworks it is common to configure your environment using XML or other text formats.

The downside of that approach is that you will have to write a lot of rather verbose XML configuration, and either you use specific tools that understand the XML dialog, or only you will discover any problems in your configuration at runtime. Lift's approach is different. Lift's configuration is pure Scala code. That means your code editor will highlight the code and the Scala compiler will find any syntactic errors at compile time. Cool, eh?

Getting ready

We use the example application we introduced in the previous recipe to walk you through a working Boot class example. Please make sure you open this project in your editor of choice to follow along. We encourage you to consciously walk through the code example as you read about the different settings. It will carve the details you learn deeper into your memory, and you will find it easier to apply that knowledge in the future.

How to do it...

When Lift starts up, it looks for a `bootstrap.liftweb.Boot` class with a `boot` method. Lift finds this class at startup by looking into a predefined package. There are ways to point Lift to a different Boot class, which might be of interest if you have different environments that you want to configure through different Boot classes. However, it is rarely necessary to change the default location, so we keep it simple and don't do that as well.

You will find the `Boot.scala` file at `src/main/scala/bootstrap/liftweb`. The path and the name of the Boot class are important. If there is no urgent need to change these defaults, just leave them as they are, it makes collaborating on a common code base easier if the expected defaults match.

The example project comes with a working Boot configuration with sensible defaults. Our configuration is extended to be used throughout this example application. Let's look at a few highlights in the code and discuss them afterwards. We have removed the comments from the shown code because of the subsequent explanation; however, the code in the project contains comments.

```
class Boot {
  def boot {

    // Set up a database connection
    if (!DB.jndiJdbcConnAvailable_?) {
      val vendor = new StandardDBVendor(Props.get("db.driver")
          openOr "org.h2.Driver",Props.get("db.url") openOr
          "jdbc:h2:lift_proto.db;AUTO_SERVER=TRUE",
          Props.get("db.user"), Props.get("db.password"))
      LiftRules.unloadHooks.
                append(vendor.closeAllConnections_! _)
      DB.defineConnectionManager(DefaultConnectionIdentifier,
                                 vendor)
    }

    Schemifier.schemify(true, Schemifier.infoF _,
                              User, UserPost)
    LiftRules.addToPackages("code")
    import BootHelpers._
    // Build SiteMap
    def sitemap = SiteMap(
      Menu.i("Home") / "index" >> User.AddUserMenusAfter >>
      LocGroup("main"),
      ...
    )

    def sitemapMutators = User.sitemapMutator
```

```
      LiftRules.setSiteMapFunc(() => sitemapMutators(sitemap))
      LiftRules.jsArtifacts =
                net.liftweb.http.js.jquery.JQuery14Artifacts
      LiftRules.ajaxStart = Full(() =>
        LiftRules.jsArtifacts.show("ajax-loader").cmd)
      LiftRules.ajaxEnd = Full(() =>
        LiftRules.jsArtifacts.hide("ajax-loader").cmd)
      LiftRules.early.append(_.setCharacterEncoding("UTF-8"))
      LiftRules.loggedInTest = Full(() => User.loggedIn_?)
      LiftRules.htmlProperties.default.set((r: Req) =>
        new Html5Properties(r.userAgent))
      S.addAround(DB.buildLoanWrapper)
    }
  }
  object BootHelpers {
    val loggedIn = If(() => User.loggedIn_?, () =>
        RedirectResponse("/user_mgt/login"))
  }
```

That's all; no hidden XML files.

How it works...

Let's walk through the code step by step.

The `boot` method starts with setting up a database connection.

```
  if (!DB.jndiJdbcConnAvailable_?) {
        val vendor = new StandardDBVendor(Props.get("db.driver")
            openOr "org.h2.Driver",Props.get("db.url") openOr
            "jdbc:h2:lift_proto.db;AUTO_SERVER=TRUE",
            Props.get("db.user"), Props.get("db.password"))
        LiftRules.unloadHooks.
                  append(vendor.closeAllConnections_! _)
        DB.defineConnectionManager(DefaultConnectionIdentifier,
                                   vendor)
      }

      Schemifier.schemify(true, Schemifier.infoF _,
                                User, UserPost)
```

`DB.jndiJdbcConnAvailable_?` checks if **JNDI** (**Java Naming and Directory Interface**—Java's implementation of LDAP) settings are available. If the servlet container has not been configured with these settings, then Lift will create a connection for you. It will read the connection settings from a property file (see below for finding the right name of the property file), or if the given property keys are not found in a property file, it defaults to using the H2 database.

Some of the terms such as "Jndi" or "servlet container" might be unfamiliar to you. While this is not the place to explain these technologies, let's just briefly describe what they do. A **servlet container** is like a runtime environment that will execute your application that complies to Java's servlet specification. Basically, when your Lift application is packaged up it creates a **WAR** (**Web ARchive**) file, which you then just drop into the servlet container's web app folder to serve it. Popular open source containers are Jetty or Tomcat.

`Jndi` is a directory (if you know LDAP, this is Java's version of it) service that can be used to store database or other access information. Your application would then point to the keys in that directory with which the actual values are referenced. It's a way to extract configuration data out of your application into the running container. On the other hand, if you have never heard of `Jndi`, there's no need to use it. It's supported, but not mandatory to use.

The next line relieves you from a whole lot of work, keeping your object model and your relational model in sync:

```
Schemifier.schemify(true, Schemifier.infoF _, User, UserPost)
```

If you use Lift's object relational mapping, Mapper, you can specify this one line to let Mapper take over the job of keeping your code and the database in sync. In this example, `User` and `UserPost` are the only model classes that we need to persist in the database. You can add here all the model classes that you need to create a database model for.

Next you need to specify the packages that Lift should scan for code.

```
LiftRules.addToPackages("code")
```

The default package name is just `"code"`, but of course you can put your application's code in a package structure such as `com.mycompany.awesomeapp`. Underneath the package that you specify here, Lift expects the `"model"`, `"snippet"`, `"lib"`, `"comet"`, and `"view"` packages.

The following block of code builds the SiteMap:

```
def sitemap = SiteMap(
      Menu.i("Home") / "index" >> User.AddUserMenusAfter >>
      LocGroup("main"),
      ...
    )

    def sitemapMutators = User.sitemapMutator
    LiftRules.setSiteMapFunc(() => sitemapMutators(sitemap))
```

Lift's SiteMap is a security feature; on the one hand, it allows you to define pages and directories, from which pages might be accessed along with the permissions the user must have in order to see these pages. On the other hand, SiteMap defines a menu structure that you can use to automatically build menus for your site. The menu you see in the example app has been built automatically through the SiteMap. We won't go into detail here; there are several tasks coming up on SiteMap.

```
LiftRules.jsArtifacts =
          net.liftweb.http.js.jquery.JQuery14Artifacts
 LiftRules.ajaxStart = Full(() =>
   LiftRules.jsArtifacts.show("ajax-loader").cmd)
 LiftRules.ajaxEnd = Full(() =>
   LiftRules.jsArtifacts.hide("ajax-loader").cmd)
```

With `jsArtifacts` we define jQuery 1.4 as JavaScript library used by default if you do not specify a different version, which you totally can.

`ajaxStart` and `ajaxEnd` define some JavaScript commands to be executed when there are Ajax requests in progress. The default setting shows a rotating spinner when the Ajax request starts and hides it when it finishes. The default commands specified here would use the jQuery 1.4 library from preceding code to generate the actual JavaScript code. You can of course modify the default behavior and, for instance, replace it with a **Loading...** message sliding down from the top of the page.

```
LiftRules.early.append(_.setCharacterEncoding("UTF-8"))
```

This tells Lift to use UTF-8 as the encoding for your templates, which is a good choice, especially if you're working with an international team or with people developing on different platforms.

```
LiftRules.loggedInTest = Full(() => User.loggedIn_?)
```

The `loggedInTest` property defines a way for Lift to check whether a user is logged in or not. We might use the Lift-provided template user for our examples, but you are not limited to using it. So with this property, you create a bridge between Lift and your login mechanism of choice.

```
LiftRules.htmlProperties.default.set((r: Req) =>
      new Html5Properties(r.userAgent))
```

With `htmlProperties` we tell Lift to serve HTML5 pages. Lift can serve XHTML or HTML5. For the remaining projects, HTML5 should be the page format of choice.

```
S.addAround(DB.buildLoanWrapper)
```

Finally, to add a DB transaction around the whole HTTP request, the line given next is added to the configuration (see the end of the boot method).

After the `boot` method we define a little helper object, `BootHelpers`. It's a place to factor out helper functions from the `boot` method itself to keep it short. Here we define a small **LocParam** (**Location Parameter**) that basically restricts access to certain pages only to logged-in users.

```
val loggedIn = If(() => User.loggedIn_?, () =>
     RedirectResponse("/user_mgt/login"))
```

There's more...

Lift supports standard key/value property resource files. Of course you can load your own files and also name them the way you like. However, Lift provides a built-in way to load the right file in the right environment. That's helpful if you are developing in a team and you have different database settings. The name of these files ends with `.props`. `.props` files are served from `src/main/resources/props`. The RunMode, username, and hostname are used to determine the correct name. We show you a few common name patterns; a full explanation can be found at the end of the article at `http://www.assembla.com/spaces/liftweb/wiki/Logging`.

PROPS filenames are dependent on the RunMode of your application. The RunMode is something like Development, Test, or Production. The username and hostname parts are optional, and the development mode can be omitted.

The default property file in development mode would be `default.props`; for production it is `production.default.props`. The property file for a developer named Henry on a machine called sparky would be `henry.sparky.props`. Henry can have different settings than other developers, and these can even differ on a machine-to-machine basis. You could use this same naming convention to integrate a logging framework such as Logback. The article at the link that we mentioned before explains how to integrate just that.

This is just a small glimpse into the abundant configuration possibilities that the Boot method offers you. One reason for its flexibility is the simple fact that it's just the Scala code. There's no XML specification it needs to adhere to. You can plug in everything that the Scala compiler understands. For instance, if you want to execute some service jobs prior to the start of the application, you can plug them in Boot. REST APIs that your application provides are plugged in Boot.

We found the best way of learning about the possibilities provided by Boot is to actually look at other existing applications and learn from them. Also, the Lift group at `https://groups.google.com/group/liftweb` will answer your questions.

Designer friendly templates (Simple)

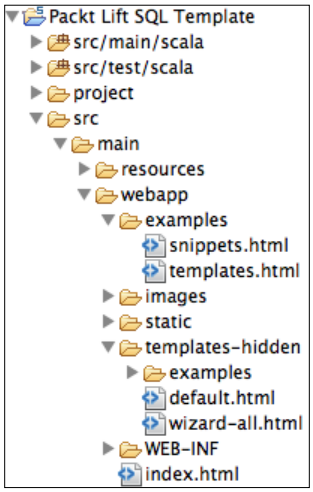
Inherent to web applications is this breach in technology. We need to combine business logic on the server with HTML pages and JavaScript on the client side. The nicely encapsulated server-side business logic then hits a client-side technology that really was intended to structure pages of text.

You somehow need to weave the backend functionality into these web pages. Countless approaches exist that try to bridge the two. Lift is also unique in this regard in that it lets you create valid HTML5 or XHTML templates that contain absolutely no business functionality, yet it manages to combine the two in an inspiring and clear way.

Getting ready

Again, we will use the example application from the *Preparing your development environment (Simple)* recipe to talk about the different concepts.

You will find the templates under the `webapp` directory inside `src/main`. If you open them, you will see they're plain and simple HTML files. It's easy for designers to edit them with the tools they know.



How to do it...

Lift's page templates are valid XHTML or HTML5 documents that are parsed and treated as NodeSeq documents (XML, basically) until served to the browser.

The standard path for everything webby is `src/main/webapp` inside your project. Say you enter a URL `liftapp.com/examples/templates` and provide the user with access to this page (see the SiteMap task for details), Lift will search the `templates.html` page inside the `examples` directory located at `src/main/webapp`. That's the normal case. Of course you can rewrite URLs and point to something entirely different, but let's now consider the common case.

Let's look at a simple template for the example applications' home page, `http://localhost:8080`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8"
          http-equiv="content-type" ></meta>
    <title>Home</title>
  </head>
  <body class="lift:content_id=main">
    <div id="main"
         data-lift="surround?with=default;at=content">
      <h2>Welcome to your project!</h2>
      <p>
        <span data-lift="helloWorld.howdy">
          Welcome to your Lift app at
          <span id="time">Time goes here</span>
        </span>
      </p>
    </div>
  </body>
</html>
```

Granted, this page doesn't do much, but that's all there is to this page.

In most applications you have some common parts on a page and some that change content. It's easy to define these hierarchies of templates. In your page template you define by which parent template you want it to be surrounded with and at which place. The parent template itself can also be surrounded by another template, and so on. This is a useful feature to extract common parts of a page into base templates and build on top of these to finally define the structure and surrounding chrome of your pages.

The parent template for this page is called `default.html` and is searched for in the `templates-hidden` folder. Any file that is embedded into a page is searched underneath `templates-hidden`. We omit the CSS and some of the Boilerplate and just show the interesting parts of the parent template's content:

```
<body>
 <div class="container">
  ...
```

```
    <div class="column span-6 colborder sidebar">
     <hr class="space" >
     <span data-lift="Menu.builder?group=main"></span>
     <hr class="space" >
     <span data-lift="Menu.builder?group=examples"></span>
     <hr class="space" >
     <span data-lift="Menu.builder?group=PostingUsers"></span>
     <div data-lift="Msgs?showAll=true"></"></"></div>
     <hr class="space" >
    </div>

    <div class="column span-17 last">
     <div id="content">The main content goes here</div>
    </div>

    ...

  </body>
```

This template defines a sidebar and places our menus there. It defines a place where messages are shown that are sent from Lift with its `S.notice`, `S.warning`, and `S.error` methods. And finally, it defines an ID (`content`) that marks the element receiving the page content.

How it works...

Let's walk through the code snippet given in the preceding section and see how the pieces fit together.

```
<body class="lift:content_id=main">
```

In the page template we tell Lift where the template actually starts. You can create complete, valid HTML pages and then make Lift cut the central piece out for its rendering process, and your designers can still work with complete pages that they can process in isolation from the rest. This line tells Lift that the content starts with the element with the ID, `main`.

The next thing we do is to define a parent template that we use to surround the page with. This way, we define essential page layout markup only once and include it everywhere it's needed. Here's how you surround a page with a parent template:

```
<div id="main" data-lift="lift:surround?
                          with=default;at=content">
… your content here…
</div>
```

In the class attribute of the `div` element you call the `surround` snippet and hand it over the `with=default` and `at=content` parameters. The `surround` snippet now knows that it should find a template called `default.html` and insert the content of this `div` element into the parent template at the point defined by the ID, `content`. Speaking of snippets, it is a mechanism to process parts of your HTML files the same way for built-in snippets as it is for your own. Snippets are pieces of logic that get weaved into the markup. We'll get to this integral part of Lift development really soon.

Lift templates are the files that are not defined in the SiteMap. They are located at a subfolder called `templates-hidden`. They cannot be accessed directly from the URL, but only through code by directly opening it or through the surround-and-embed mechanisms inside other templates or pages.

Have a look at the parent template `default.html` shown previously. This file, along with the other files we discuss here, is available in the source code that comes with the book. It's a standard HTML5 file defining some styles and finally defining a `div` element to bind the child content:

```
<div id="content">The main content will get bound here</div>
```

Lift will remove the text inside the DIV and replace it with the actual content, as shown in the following screenshot:

A few other things at the top of the template are worth noting:

```
<style class="lift:CSS.blueprint"></style>
<style class="lift:CSS.fancyType"></style>
<script id="jquery" src="/classpath/jquery.js""
        type="text/javascript"></script>
```

Lift comes bundled with the Blueprint CSS framework (`http://blueprintcss.org/`) and a version of jQuery (`http://jquery.com/`). It's intended to make it easier for you to start, but by no means are you bound to using Blueprint or the included jQuery version. Just use your own CSS framework (there's a recipe on using Twitter's Bootstrap) or jQuery where it makes sense.

For instance, to use a hosted version of the latest jQuery library, you would replace the `script` tag from the preceding code snippet with the following:

```
<script type="text/javascript" src="http://code.jquery.com/jquery-
1.8.2.min.js"></script>
```

Lift provides some standard snippets which you can use to build up your pages. The `default.html` template utilizes a snippet to render a menu and another snippet to place messages on the page:

```
<span data-lift="Menu.builder?group=main"></span>
```

When you define the element that encloses the menu, Lift will automatically render it. If you omit the `group` parameter, all menu entries will be rendered. Having that parameter will restrict the menu only to the items within that group. You can assign a menu group (called `LocGroup`) in the SiteMap you defined in the Boot class.

```
<div data-lift="Msgs?showAll=true"></div>
```

This snippet call will render messages that are produced by the backend application in this spot.

There's more...

We will now have a look at execution order.

In normal execution mode, Lift first evaluates the outer snippets and then layer by layer moves to the inner snippets. If you want to include the result of some inner snippet evaluations to the input of the outer snippets, you need to reverse that process. For that very reason, Lift provides a snippet parameter, `eager_eval=true`, that you add to the outer snippet:

```
<div data-lift="ImOuter?eager_eval=true">
  ...
  <div data-lift="ImInner">
    ...
```

```
    </div>
    ...
</div>
```

Adding that parameter causes Lift to first evaluate the inner snippet and then add the result of the inner snippet call to the input that is processed by the outer snippet.

You can also embed templates into your page or other templates. That's the opposite operation of surrounding a page, but equally simple. In your page, use the `embed` snippet to embed a template:

```
<div data-lift="embed?what=/examples/templates/awesome"></div>
```

The `what` parameter defines the path to the template, which is searched for within the `webapp` directory.

We will now see the programmatic embedding of templates.

You can easily search a template and process it programmatically. In that case you need to specify the `templates-hidden` directory; that way you are able to access top-level pages as well.

```
val ns:Box[NodeSeq] = S.runTemplate(List("templates-hidden",
"examples", "templates", "awesome"))
```

Please see the `EmbedTemplates` snippet for an example of how to programmatically access templates and apply transformations before embedding it.

```
<div data-lift="EmbedTemplate?what=/examples/templates/awesome"></div>
```

As you can see, our own templates are called just the same way as Lift's default templates, and they can do the same things.

Programmatic access to templates is useful, for instance when you want to send HTML e-mails. Inside the mail sender you would grab the template, process it (see CSS Selectors), and send the complete HTML to the recipient.

There are a myriad more reasons or use cases when you want to access your templates from your Scala code. Just keep in the back of your mind that you can do it.

The `S.runTemplate` method will fetch the template and process it. That means it will look for any embedded Lift snippet calls and execute them. These snippet calls could potentially embed other templates recursively.

If you do not want the template to be processed, you can retrieve it like this:

```
val tpl:Box[NodeSeq] = Templates(List("templates-hidden", "examples",
"templates", "awesome")
```

Lift templates are very powerful, and they have to be. They are at the basis of every web application and need to handle a lot of different scenarios.

The separation between the markup and the logic keeps the templates clean and prohibits your designers from breaking code. It might take a while to adopt to this template style if you come from a framework that mixes markup and code. We believe, especially in larger applications, you will soon see the benefits of a clear separation and encapsulation of your logic in reusable pieces. Speaking of reusable pieces, let's head over to snippets, Lift's way to plug functionality into templates.

The Lift wiki offers further information about templates and binding at the following links:

- `http://www.assembla.com/spaces/liftweb/wiki/Designer_Friendly_Templates`
- `http://www.assembla.com/spaces/liftweb/wiki/Templates_and_Binding`

Using Lift snippets (Simple)

Every web application that does more than rendering static content needs some way to add logic to the pages it sends to the browser. Since Lift does not allow any logic inside its templates (see the previous recipe), there must be a different mechanism. In Lift these logic parts that plug into the page are called **snippets**.

Getting ready

In the previous recipe we have shed some light on Lift's template mechanism. The templates are the user interface of your application. Now we need to discuss how we can bind logic to it. Lift uses a different approach than most other web frameworks. Lift calls this approach "View First". We'll discuss what it means and why we think it's better suited for this kind of application development. We'll show you different forms of snippets and how you can develop your own.

You will find a `snippets.html` page in the example application that we use to showcase the different forms of snippets.

How to do it...

A common pattern to connect the user interface with the backend logic is called Model-View-Controller. This pattern is used in most web application frameworks. It tries to separate your business model from the user interface (separation of concern) by putting a controlling mechanism in between, which mediates between the backend (the model) and the view.

These frameworks put the controller first. A certain URI (`/user/show/123`) triggers a controller that is bound to that URI. That controller is the important one that handles calls to the backend and finally puts results into the page.

Lift's approach is different. In Lift, the view comes first. A URI is bound to a specific page. That page then usually defines a number of logic parts that are more or less distinct from each other. A page usually has a menu; some pages have a shopping basket, or other functional pieces that make up the page. We believe this approach is better suited to the nature of web pages. If you want to use the same functionality for a different page, no problem, just take the snippet and put it into that other page. The Lift wiki presents a much more thorough introduction to View First at the following link:

`http://www.assembla.com/wiki/show/liftweb/View_First`

In the previous recipe you learned how to specify a snippet inside a template. All you need to do is to add some markup to an element:

```
<div data-lift="MySnippet.myMethod">
…
</div>
```

There are several mechanisms to reference snippets from a template; the newest one is using `data-lift`. Alternatives are discussed ahead.

Now we create the snippet class or object that we just referenced:

```
class MySnippet {
  def myMethod =
    ".current-time *" #> now
}
```

That's the basics of snippet reference and invocation. Let's look into this more closely.

How it works...

Lift provides you with the following ways to reference a snippet:

- `class="lift:MySnippet.myMethod"`: Specify the snippet and, optionally, a method to call inside the class attribute of an element. Prefix that snippet name with `lift:`.
- `class="l:MySnippet.myMethod"`: This is the same as the preceding one, but a prefix of `l:` is enough.
- `data-lift="MySnippet.myMethod"`: Since Lift 2.4 you can specify an HTML5 compliant attribue, `data-lift`, to hold your snippet call. No prefix is required.

If you do not give a method name, then Lift assumes that the method to call inside the snippet is `render`. Optionally, if your snippet supports it, you can hand over parameters to the snippet, as follows:

```
<div data-lift="MySnippet?param1=123;param2=789"></div>
```

Snippets are looked up in the `"snippet"` subpackage of one of the packages that you added in Boot. So for instance, if you added `"code"` as your source package (`LiftRules.addToPackages("code")`), then `"snippet"` is expected to be a child package of `"code"`.

Now, what does the snippet process? The element that contains the snippet call along with all its children is passed to the snippet call as input. The data type for that is a `NodeSeq` (a sequence of XML elements). The snippet processes this `NodeSeq` input and returns another `NodeSeq`, which replaces the original content. So, your snippet can do whatever it wants to with the content. It can enhance it, replace it, add another template, or return an empty content if that element should not be visible to the user. Please note that this is a very oversimplified perspective on how snippets work. You can do all these things in many different ways. But in the end a snippet takes the template XML, wraps it, and returns a processed version of it.

Let's look at a minimal snippet example:

```
class TimeSnippetClass {
  def render: CssSel =
    ".current-time *" #> now
}
```

That's a valid snippet which, granted, doesn't do much. A snippet is basically either a class or an object that defines a bunch of methods. A snippet can have more than one transformation function.

If the function's name is `render`, then you can omit its name in the snippet template binding.

There are a few valid method signatures for these methods. The one you saw just now returns a bunch of CSS selectors (please see the next recipe on CSS selectors) of type `net.liftweb.util.CssSel`. Lift then applies the templates to these functions to produce the resulting `NodeSeq` output. Another option is a function that takes `NodeSeq` as input and returns an output `NodeSeq`:

```
def render(in: NodeSeq): NodeSeq = {
  val cssSel = ".current-time *" #> now
  if (number > 500) cssSel(in) else NodeSeq.Empty
}
```

`cssSel(in)` applies the input XML to the CSS selector function and returns the resulting XML. If, however, that random number is smaller or equal to 500, the function will return empty, effectively stripping the input XML from the page.

If you define a snippet as a class, it will be instantiated by Lift on a per request basis. That means that all calls to a certain snippet for one request and subsequent Ajax requests will go to one and the same instance of the snippet class. Other requests will access their own instance of the snippet. That in turn means it's safe to store values as instance variables of the class.

Not the same, however, for objects! While it's a common pattern to create snippets as objects, make sure you never store request-related information on the object level. Objects are singletons: only one instance is created per application. So every value you save on the object level is seen by every request. For user passwords, that would be disastrous. If you keep data inside a method, though, it's perfectly safe. Method variables are locally scoped and not visible to other calls. But that also means you cannot easily share this information.

It's a common use case to share some information between snippets on a per request or even per session basis. A **per request basis** means that the information is created with a new request and will be available for subsequent Ajax requests. The HTTP request shown next would wipe the existing information and create a new one. A **per session basis** means that the information is created with the session (for instance when the user logs in) and destroyed when he logs off.

Lift provides a type-safe and easy-to-use way to create this kind of information. For an example in the code, please see the `VarExample` snippet and its usage in `snippets.html`:

```
object VarExample {
  object exampleRequestVar extends RequestVar[Int]
         (randomInt(1000))
  object exampleSessionVar extends SessionVar[Int]
         (randomInt(1000))
  def render =
    ".request-var *" #> exampleRequestVar.is &
    ".session-var *" #> exampleSessionVar.is

}
```

You define a request scoped variable as follows:

```
object myRequestVar extends RequestVar[Int](0)
```

A request variable is defined as an object extending `RequestVar`. You give it the type it should hold (`Int` in this case) and initialize it with a constant value or, as in the preceding example, with a method call.

You can assign it a new value by calling the following:

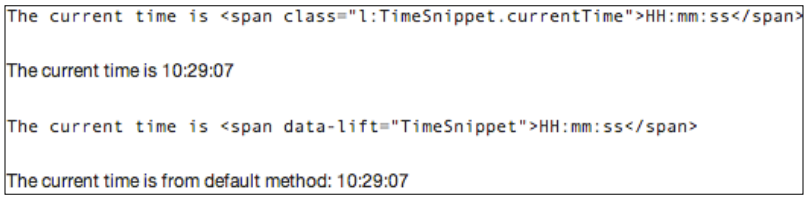
```
myRequestVar(newIntValue)
```

Or you can access its value with the following:

```
val itsValue = myRequestVar.is
```

It's just the same for `SessionVar`. There's a bunch of other specialized Vars, but these two are the most important ones.

The following screenshot demonstrates the rendering of the template containing calls to the snippet with the embedded snippet results:



There's more...

That's a whole lot of information. And yet, there's more.

It's easy to access URL parameters in snippets. URL parameters are these values after the `&` sign in a URL. In a snippet you access them with the `S.param` method:

```
val param = S.param("next") openOr ""
```

`S.param` returns `Box[String]`. A box is a wrapper that can be full (there is something in there) or empty (nothing there). Box is an extended version of Scala's option that also adds a failure state. `openOr` opens the box if something is in there or returns an empty string if the box is empty. Box and option are great ways to get rid of `null` and these ubiquitous null checks in every corner.

The following are the subtypes of snippet:

- Another subtype of snippet is `DispatchSnippet`. For an example, see `DispatchTimeSnippetClass` in the example code. `DispatchSnippet` instances are basically like normal snippet classes or objects with two differences. They extend the `DispatchSnippet` trait and need to override a `dispatch` method. This method clearly defines which methods can be called from a template and with which name. You can use this feature when you want to limit the number of methods that can be called from a template or define different names for them.
- Yet another form of snippet is `StatefulSnippet`. `StatefulSnippet` builds on top of `DispatchSnippet`, so you also need to define a `dispatch` method.

- The `StatefulSnippet` instances—there isn't really a stateless snippet, though—are kept around longer than other snippets. That is useful, for instance, if a form spans multiple request/response cycles. If you find yourself with a need to restore the state of a snippet with values it had during the previous cycle, have a look at `StatefulSnippet`. More information can be found at `http://simply.liftweb.net/index-4.3.html`.

And in case you still want to learn more about templates and snippets, *Simply Lift* contains numerous more examples and detailed information on the subject at `http://simply.liftweb.net/index-3.3.html#toc-Section-3.3`.

CSS selector bindings (Simple)

Now that you have templates on one side and snippets on the other, you need to combine the two. There needs to be some special sauce that lets you glue backend data with frontend pages. CSS selectors are this special sauce. They provide a well-known syntax to select parts of the template and bind them to values or form elements in the backend.

Getting ready

In the previous recipes you have already seen CSS selectors. You probably wondered about some strange syntax coming your way. Now is the time to explain these constructs in more detail.

We won't go into all the possible details; we will show you the most widely used cases and give you pointers where you can learn more.

How to do it...

If you know jQuery, you already have a good idea how CSS selectors work.

The idea is to select an element of your markup based on some distinctive feature of that element. So for instance, you could select by element ID, by class name, or by attribute value.

Before you use selectors in your snippet, please add the following two imports:

- `import net.liftweb.util._`
- `import Helpers._`

These imports provide implicit conversions that make the use of CSS selectors possible.

The left-hand side of a CSS selector is a string denoting that element in the page you want to grab hold of. A few examples are as follows:

- `#user-name`: It selects the field with the `user-name` ID
- `#user-name *`: It selects the children of the field with the `user-name` ID
- `.expense`: It selects all elements that have the `expense` class
- `.expense *`: It selects the children of all elements that have the `expense` class
- `name=income` or `@income`: It selects the input field with the `name` attribute set to `income`
- `#report [href]`: It selects the `href` attribute of the element with the `report` ID
- `#report [class+]`: It selects the `class` attribute of the `report` element and lets you add new classes to it
- `#report [style!]`: It selects the `style` attribute of the `report` element and lets you remove an element from it

Please see the `selectors` page in the example application for some live examples. Feel free to play with the code, modify the selectors or the applied transformations, and observe the results.

How it works...

The list given at the end of the preceding section describes some of the more often used CSS selectors. Through an implicit conversion, the left-hand side selector is converted into something of the `ToCssBindPromoter` type and the `#>` method is added.

On the right-hand side of the expression you add the transformation you want to perform on the element. These transformations range from simply adding a value from a database or some other source, to binding complete UI components, to attaching Ajax functionality to links, buttons, or other elements. You can also use these transformations to iterate over a list of input values, for example a SQL result, and create as many output elements as needed. Please see the next recipe for a deeper look into these kinds of bindings.

So let's look at a few examples (see the example code for more of them).

Suppose the following template is given:

```
<span><span id="user-name">Name Here</span></span>
```

And the following selector is given:

```
"#user-name" #> "Claudia"
```

This would result in the following:

```
<span>Claudia</span>
```

Let's use a slightly different selector:

```
"#user-name *" #> "Claudia"
```

This produces the following result:

```
<span><span id="user-name">Claudia</span></span>
```

The second selector uses `*` to select the children of an element, not the element itself. So only the children of the selected element get transformed into the right-hand side of the expression, and the element itself remains. Let's see how we can set the `src` attribute of an image for the following template:

```
<img class="author-img" src="">
".author-img [src]" #> (S.hostAndPath + "/images/userimg.jpg")
```

Here we select the `src` attribute of all elements with the `author-img` class and set its absolute path to `userimg.jpg`. Using the absolute path is not necessary here, but it's a great opportunity to point out the `S` object that provides you with a host of useful methods, such as giving back the URI to the current active page or `hostAndPath`, the hostname, optional port, and path to your web application, excluding any path to the current page. The following are the characteristics:

- Values on the right-hand side
- Combining selectors with `&`
- All selectors see the same input; they are applied in one batch, not one after the other; and each sees changed input
- CSS selectors (`CssBindFunc`) are at the core `NodeSeq => NodeSeq` functions that can be applied everywhere where `NodeSeq => NodeSeq` can be applied

There's more...

In most of the cases you will wish to combine CSS selectors and not just use one per render function. You can do this very easily using the `&` method, as follows:

```
"#user-name *" #> "Claudia" &
".record [class+]" #> "selected" &
".author-link [onclick]" #> ajaxInvoke(()=>Alert("Hi..."))
```

This snippet binds the username, adds the `selected` class to the classes of the `record` element, and binds an Ajax action to the `onclick` handler of `author-link`. Whenever someone clicks on the link, this server-side action is invoked and the resulting JavaScript is returned to the browser and executed. In this case we just show a mostly meaningless alert message in the browser. That's supposed to be a teaser to show you how easily you can add Ajax spice to your application dish. The next recipes will cover this in greater detail.

We could only touch the surface of what's possible with CSS selectors. They provide a very powerful and natural way to link templates and snippets together.

For more information, please see the Lift wiki at `http://www.assembla.com/spaces/liftweb/wiki/Binding_via_CSS_Selectors` or see the online book *Simply Lift* at `http://simply.liftweb.net/index-7.10.html`.

The text tasks deal in detail with binding dynamic content such as query results or lists of things, and Ajax functionality. They all build on top of CSS selectors to provide their functionality.

Binding dynamic content (Medium)

In your web application you want to display variable and dynamic content, for example a variable number of records read from a database, or searching results returned from some backend service, or a number of items from an Atom feed.

The mechanism for doing that has already been explained in the previous chapters, yet because this is a topic very important for the overwhelming number of applications in one form or another, we will shed some extra light on it and explain a few simple details.

Getting ready

You should have read and understood the previous tasks on designer friendly templates, snippets, and CSS selectors. Binding dynamic content is really just an application of these mechanisms to this specific problem domain. The example application contains the page "Dynamic Content", which will contain the examples on this task.

How to do it...

The example template for this can be found in the `dynamic.html` file located at `webapp/examples/task8/`.

So let's say you have the following template:

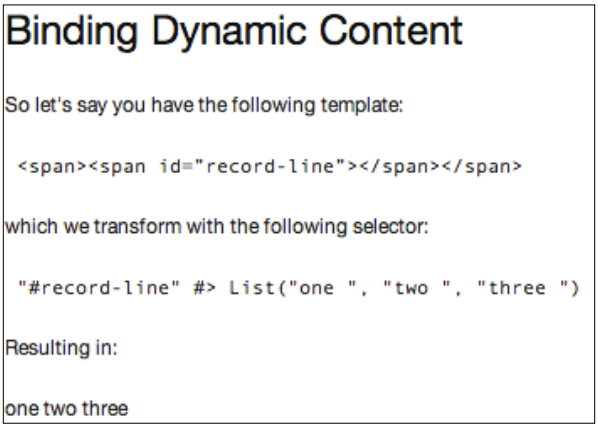
```
<div id="main" class="lift:surround?with=default;at=content">
  <h2>Binding Dynamic Content</h2>
  <p>So let's say you have the following template:</p>
  <pre><code> &lt;span>&lt;span
    id="record-line">&lt;/span>&lt;/span> </code></pre>
  <p>which we transform with the following selector:</p>
  <pre><code>
    "#record-line" #> List("one ", "two ", "three ")
  </code></pre>
  <p>
```

```
    Resulting in:
    <div data-lift="Dynamic.renderList">
      <span><span id="record-line"></span></span>
    </div>
  </p>
...
```

From that template we call the `renderList` method of the `Dynamic` snippet. This method contains the necessary selectors to transform parts of the input template into HTML output. Lift's selectors are very similar to CSS selectors; you use them to identify specific parts of your input HTML:

```
object Dynamic {
  def renderList =
    "#record-line" #> List("one ", "two ", "three ")
  ...
}
```

The resulting browser output would be:



The output HTML for this selector looks like the following code:

```
<div>
  <span>one two three </span>
</div>
```

The `span` element with the `record-line` ID has been removed and replaced with the content of the list. If you would pass in an empty list (nil), an empty box, or none, the result would be an empty outer span, as follows:

```
<span></span>
```

Now, what would it look like if you bind not to the record-line element itself but to its children? Let's see:

```
"#record-line *" #> List("one ", "two ", "three ")
```

Did you see `*` in `"#record-line *"`? That tells the selector to use the children of the selected element and not the element itself. And the result would be:

```
<span>
  <span id="record-line">one </span>
  <span>two </span>
  <span>three </span>
</span>
```

The content looks the same in the example page; the template, however, is different. Lift kept the element's surrounding span, and you may also notice one other thing. The first element keeps the ID that was in the original template, but it is removed from the other elements. The reason is that an ID has to be unique within an HTML page. Lift helps you here by stripping the duplicate IDs in order to create valid output.

Now, let's dive into the real stuff.

How it works...

Until now the right-hand side of the selector binding, the transformation rule, has merely been some constant value. However, Lift also allows `NodeSeq => NodeSeq` functions as part of the right-hand side—transformation rule. So let's assume the following template:

```
<table>
  <tr class="expense-row">
    <td class="date">YYYY/MM/DD</td>
    <td class="desc">Description<td>
  </tr>
  <tr class="clearable">
    <td class="date">YYYY/MM/DD</td>
    <td class="desc">Description<td>
  </tr>
  <tr class="clearable">
    <td class="date">YYYY/MM/DD</td>
    <td class="desc">Description<td>
  </tr>
</table>
```

We want to bind the content of one row to actual values. We can do it in the following way:

```
".expense-row *" #> (".date *" #> getExpenseDate &
        ".desc *" #> getExpenseDesc) & ClearClearable
```

Applying the template to this transformation would result in the following:

```
<table>
  <tbody>
    <tr class="expense-row">
      <td class="date">2012/06/16</td>
      <td class="desc">Description Line 1</td>
    </tr>
  </tbody>
</table>
```

The resulting template contains a `tbody` tag, which was inserted by the parser that Lift uses to read the HTML5 template to ensure that it's standard compliant. Other than that, you see how the cell values have been replaced with actual data from the snippet.

Oh, and here's one more neat trick. In the preceding template you see the `clearable` class assigned to extra rows. That's a way for designers to add fake data into a template so that you can see what it would look like if there was real data. Then in your snippet, use `ClearClearable` to remove all parts from your input template that have this class to make sure the extra data is gone.

There's more...

Now let's go one step further and make this example useful. Let's add a number of rows to the result page.

The template will stay the same, but in our snippet, we bind to a list of `NodeSeq => NodeSeq` functions:

```
".expense-row *" #> List((".date *" #> getExpenseDate &
                  ".desc *" #> getExpenseDesc),
                 (".date *" #> getExpenseDate &
                  ".desc *" #> getExpenseDesc))
```

The result will of course be two rows in our table—a big step into the map-dynamic-data direction. Combining what we have learned so far with the `map` function that we find in Scala's lists, we can do the following:

```
case class Expense(date: Date, desc: String)
def renderTableWithMap = {
  val records = Expense(getRandomDate(100), getExpenseDesc) ::
      Expense(getRandomDate(100), getExpenseDesc) ::
      Expense(getRandomDate(100), getExpenseDesc) :: Nil
  ".expense-row *" #> records.map { record =>
    ".date *" #> getExpenseDate(record.date) &
    ".desc *" #> record.desc
    }
  }
```

First we create a little case class, `Expense`, which will hold one record of data. Case classes are really cool for that.

Next, in the snippet we create a list of two entries. We simulate our very awesome database routine that had returned three records. `records.map` walks through each element of this list and creates a transformation rule for each element. This list of transformation rules is then applied to `.expense-row *`. The result of this snippet is the same as the previous one, where we applied the list of transformations directly. However, this notation lends itself much better to transformations over dynamic lists of data and this is the syntax you should use.

What if you would like to bind something different if you have zero records found? You could change the snippet to something like the following code snippet (see `renderTableWithMapConditionally`):

```
if (itsRecords.size > 0) {
  ".expense-row *" #> records.map { record =>
    ".date *" #> getExpenseDate(record.date) &
    ".desc *" #> record.desc
  }
} else {
  ".expense-row *" #> <td></td><td>No records found</td>
}
```

Since you always have to return a transformation rule, you cannot just put an `if` statement around your transformation and omit the `else` branch. Instead you have to supply something meaningful there. In this example we just construct some inline XML that we return.

Another thing that is actually very useful from time to time is to put a selector there that does nothing:

```
if (itsRecords.size > 0) {
  ...
} else {
  "#notExistentSelector" #> ""
}
```

You can create `val notExistent = "#notExistentSelector" #>` and reference that when needed.

In this task we faked the dynamic data by putting it into the snippets. In the upcoming tasks on using MongoDB together with Lift, we will present some examples that actually retrieve data from the database and then present it in the page.

Managing page access (Simple)

I know, security. As a developer you want to create great content and not have to worry about bad people who try to break into your creations.

One part of securing your application is to restrict access to the different parts of your website depending on who's standing at the door. You want to have the home page open for everyone, yet your application's functionality should only be available to registered users, and some additional admin functionality should only be accessible by very few people.

Lift offers a built-in security gateway called **SiteMap**. You don't need to check access on every page but rather define who can do what in one central place. In the *Saying hello to Lift boot (Simple)* recipe when we discussed the Boot class, you already saw that definition; now we're going to look at some of the functionality the SiteMap offers in greater detail.

Getting ready

Lift's SiteMap configuration is a part of the Boot class, Lift's startup mechanism. All the different settings we discuss here can be found in this class. In addition we will create some pages and snippets that demonstrate the different use cases. We'll point to them on the way. Make sure to study the code as it's the code that won't lie to you: either it's working or it's not.

How to do it...

To start securing your site you create a SiteMap function using the SiteMap object's `apply` method. This function is created in the Boot class of your project. Let's look at one of these, which we use in the example project. Perform the following steps:

1. First, create a helper function that we need later to check if a user has logged in:

```
val loggedIn = If(() => User.loggedIn_?, () => RedirectResponse("/
user_mgt/login"))
```

2. Next, define the SiteMap:

```
def sitemap: SiteMap = SiteMap(
```

3. Define a `"Home"` menu entry:

```
  Menu.i("Home") / "index" >> User.AddUserMenusAfter
         >> LocGroup("main"),
```

4. Define some additional entries:

```
  Menu.i("4 - Templates") / "examples" / "task4" / "templates"
         >> LocGroup("examples")
         >> Title(i=>Text("Templates Task"))
         >> loggedIn,
```

```
    Menu("Static", "Static Content") / "static" / **
           >> LocGroup("main") >> loggedIn,

    // Omitted additional definitions ...
    Menu.i("17 - REST Example") / "examples" / "task17" / "rest"
           >> LocGroup("examples") >> loggedIn
)
```

5. Hook the `User` object into the menu structure:

```
def sitemapMutators = User.sitemapMutator
```

6. Set the SiteMap:

```
// set the sitemap.  Note if you don't want access control for
// each page, just comment this line out.
LiftRules.setSiteMapFunc(() => sitemapMutators(sitemap))
```

   The menu entries you will enter in your own project will of course vary, yet it will be of the same style.

7. Now add the following markup to the page template in `templates-hidden/default.html` to display the main menu at that place:

```
<span class="lift:Menu.builder?group=main"></span>
```

How it works...

Let's see what we did here.

We use SiteMap's `apply` function, feeding it with a number of menu entries, to create an access control list of sorts. SiteMap is a bit like a central control for the different pages of your site. It allows you to define menu entries, access control, and also attach additional data to the single menu items that you can use later.

In step 3 we defined our first menu entry. Let's build it up bit by bit:

```
Menu.i("Home") / "index"
```

This will create a menu entry `"Home"` pointing at the `index.html` file. As you see, you omit the suffix of template files.

If your menu entry should have a different text appearing in the menu, maybe because you localize it, you can write the following line of code:

```
Menu("Home", S ? "Home") / "index"
```

The name of the menu item will stay `"Home"`, and the text shown in the menu will be localized depending on the browser settings of the visiting user.

We can add parameters to menu entry called LocParams. We use this feature to tell Lift to add the user menu after the `"Home"` item, and we also set a menu group, a way to separate menu items from each other so that you can later show them in different places:

```
Menu.i("Home") / "index" >> User.AddUserMenusAfter >> LocGroup("main")
```

There isn't much security yet, as we are allowing any user to access this page. Let's see how we can put a page behind access control. First we define a LocParam that checks if the user has logged in, which we did with defining the `loggedIn` LocParam.

The `If` LocParam takes a function that checks if a user has logged in returning `true` or `false`; the second parameter is a function returning a LiftResponse, which is called for users not logged in. In our case we simply redirect them to the login page. Now let's apply this check.

If you look at the second menu item definition in our example, you will be able to see how this parameter is applied.

The menu item **4 – Templates** is only accessible to authenticated users. After the same pattern you could create a `isSuperUser` LocParam that would additionally check the role or extended permissions of a user and would only return `true` if all criteria are satisfied.

"So, do I have to put every page I ever want to serve into the SiteMap?", you may ask. Nope. Lift provides a way to define access for one or several directory hierarchies. One example is the definition of the static menu item. The `**` method at the end of that definition says that this definition is applicable for every item below the `static` directory, even for subdirectories. If you only want to grant access to one directory level, use `*` instead.

After you have defined your SiteMap you publish it to Lift via `LiftRules.setSiteMapFunc`, a method that takes a function argument, which in turn will emit a SiteMap, as shown in step 6.

In step 5 in the preceding section, we massage the defined SiteMap a little bit by piping it through `sitemapMutator`. `sitemapMutator` takes a SiteMap as an argument and returns a potentially modified version of the SiteMap. Since our `User` object defines menu entries (`Login`, `Register`, `Edit`, and so on), we need to add them to our custom SiteMap somehow. We do this in two steps. Step one is a definition in our SiteMap that tells the mutator where the menu should be placed:

```
Menu.i("Home") / "index" >> User.AddUserMenusAfter
```

`user.AddUserMenusAfter` is kind of a marker that says that the `User` object's menu entries should be placed after the `"Home"` menu entry. Other options would be `AddUserMenusUnder` to add the entries as a child menu of the current menu item, or `AddUserMenusHere` to replace the current menu item with the `User` object's menu items.

To display the menu, we need to add the `Menu` snippet to a page, preferably to some base template such as `templates-hidden/default.html`. You see there how the menus are split up and served separately.

Step 7 shows the markup that needs to be added to an HTML page. We add a group definition to the snippet call, so this will only render menu entries that belong to the `main` LocGroup.

There's more...

We'd like to show you one more use case for SiteMap and menu parameters, parsing URL parameters into snippets.

Lift provides a way to parse URL parameters in URLs such as `/app/show/1234` into type-safe representations without the need to rewrite the URL. The example application contains the `examples/show` and `examples/show2` pages to demonstrate this case using classes and objects. In this section we will discuss injecting URL parameters into snippet classes.

First of all, let's define a case class to hold our data. We put the definition into the Boot file; in real applications there are surely better ways to stash it.

```
case class DocumentInfo(docId: String)
```

Next, define `Menu.param` for the URL that should contain the parameter:

```
Menu.param[DocumentInfo]("ShowDoc2", "Show Document",
s => Full(DocumentInfo(s)),
dinfo => dinfo.docId) / "examples" / "show2"
>> LocGroup("main") >> loggedIn >> Hidden
```

`Menu.param` is typed with the case class that will store our data. We give it a name (`ShowDoc2`) and a link text (`Show Document`), although that text will not show up because we will hide this menu entry later with `>> Hidden`. Now we define two functions: the first one takes a string argument and constructs our case class. We could parse the data here, split it up, and so on. If the parameter is invalid, we would return an empty box instead. The second function takes the case class and extracts the parameter out of it. The rest of the definition is in line with the other examples we have looked at.

Now, in order to use the parameter, you define a snippet class that takes the case class as constructor parameter and then uses that parameter in the `render` function:

```
class ParamInjectClass(docInfo: DocumentInfo) {
  def render() = "*" #> docInfo.docId

}
```

If Lift cannot find the `DocumentInfo` instance, it will not instantiate the `snippet` class.

Using menu parameters with a snippet object is slightly different. For an example, please see `ParamInjectObject` in the example code base.

Lift's SiteMap is awfully flexible. And as with most other places we just scratched the surface. We believe, though, that we have covered the large number of scenarios that you run across in small or medium applications. Understanding the LocParams can also be a bit tricky, so if you have questions, don't hesitate to consult the friendly Liftafarians over at `http://groups.google.com/group/liftweb`.

Building a dynamic menu structure (Advanced)

The previous recipe has shown us how we can create a menu structure with Lift's SiteMap. We have also briefly looked at how we can parse URL parameters and pass them to Lift snippets in a type-safe fashion.

Building upon this knowledge, this recipe will introduce you to dynamic menus: menus that change with the current page content.

Getting ready

Most probably you have seen this kind of scheme in web applications near you. You log in to a site and the URL of the application becomes `http://www.app.com/.../user/123`, and it will show you information about your user. Then, maybe if it's a photo service, you can drill down into a specific photo album and the URL becomes `.../user/123/album/789`, and it shows you photos from that album.

The suggested way in Lift to handle these kinds of situations is the use of `Menu.param`. Don't be afraid if you don't understand everything after one read—sometimes you need to come back later and revisit what you've seen to truly understand.

This recipe will introduce you to a `Menu.param` example. The code for it can be found mostly in `UserPosts.scala`; the menu entry in the example application is `User List`. Make sure to study the code along the way.

In our example application we want to list the users who have a user account. Clicking on a user will lead you to another page with a list of posts of that user. Clicking on one post will show you the contents of it. In addition, if you click on the user who is currently logged in, you will see a simple form to enter a post. You will need to use this to populate the database in order to see some post listings. The URL for showing such a post will be `http://localhost:8080/examples/users/1/posts/1`.

In addition to the `User` object, we create another database object called `UserPost`, which will store the data of one post and associate it with the authoring user.

How to do it...

To build a dynamic menu structure perform the following steps:

1. To begin implementing this mechanism we start in Boot with the definition of a `User List` menu item:

```
MainUserPostsPage.menu >> loggedIn,
User.listUsersMenu >> loggedIn,
AUserPost.menu >> loggedIn
```

2. Drawing on what we learned in the previous recipe we create a menu item in the `MainUserPostsPage` object in `UserPosts.scala`:

```
object MainUserPostsPage {
  lazy val menu = Menu.i("User List") / "examples" / "users"
                  >> LocGroup("examples")
```

3. The user's page itself is pretty simple; it will just use this object's `render` method to list all users (`UserPosts.scala`):

```
def render = "li *" #> User.findAll.zipWithIndex.map{
  case (user, idx) =>
    "a *+" #> (idx + ": " + user.shortName) &
    "a [href]" #> User.listUsersMenu.calcHref(user)
}
```

4. The user menu is defined in the `User` object:

```
lazy val listUsersMenu = Menu.param[User]("AUser",
     Loc.LinkText(u => Text(u.shortName)),
                  id => User.find(id),
     (user: User) => user.id.is.toString) / "examples" /
     "users" / * >> LocGroup("PostingUsers")
```

5. Let's create the `listUsers` snippet that will display all posts of one selected user (`UserPosts.scala`):

```
def listUsers = {
  ".back-to-users [href]" #>
     MainUserPostsPage.menu.loc.calcDefaultHref &
  "li *" #> UserPost.findAll(By(UserPost.userId,
                    user.id.is)).map(post =>
  "a *+" #> post.title &
  "a [href]" #> AUserPost.menu.calcHref(user -> post))
}
```

6. The final piece is the `AUserPost` object that provides the menu item for a user's post (`UserPosts.scala`):

```
lazy val menu = Menu.params[(User, UserPost)]("AUserPost",
     Loc.LinkText(tpl => Text("Post: "+tpl._2.title)),
     {
       case User(u) :: UserPost(up) :: Nil => Full(u -> up)
       case _ => Empty
     },
     (tpl: (User, UserPost)) => tpl._1.id.is.toString ::
      tpl._2.id.is.toString :: Nil) / "examples" /
      "users" / * / "posts" / * >> LocGroup("PostingUsers")
```

How it works...

In step 1 we hook the menu items we need into the SiteMap. In addition to the `MainUserPosts` menu, we already define the menu item to show a list of posts for a user (`/examples/users/1`) and for the post detail page (`/examples/users/1/posts/1`). We explain these later, but now you have all Boot changes in one place.

We define a menu item in step 2 pointing to `/examples/users.html` and move it into the `examples` group so that it will appear in the correct menu.

To show a list of all users, we basically iterate in step 3 over all users and create a new entry in that `<li>` list element. A neat trick is to use `zipWithIndex` to bind an index to each user, which we then use to enumerate the user list. We facilitate the `User` object's menu (`listUsersMenu`) to create the correct link for that user. Clicking on one of these user links would lead to that user's page (`/examples/users/1`).

In step 4 we define a menu item for a user, passing in that user `u`. We extract the name to show up in the menu item, and given the ID of the user we query the database with `User.find`. The next function tells the menu item how the `User` object is transformed into some string ID with which the user can be uniquely identified. Since all our database objects have unique IDs, you can use just that. You could also map the database ID to a random string to not hand over database IDs to the client.

We need to create a way to display the list of posts for one user, which we do in step 5.

Now we can link to one user's page, but what's the name of that page in our page structure? The menu param shows `*` for all pages underneath the `users` folder. So the name of the page is simply `star.html`. `/examples/users/star.html` is the page that lists all posts for one user facilitating the `PostingUser.listUsers` snippet.

We select a list of all posts of that user and bind the post's title as well as a link to that specific post. To calculate the link we pass in a tuple of the current user and post instance.

In step 6 the menu item is passed in a tuple of user and post. It extracts the link text from the title of the post. Next we use the `unapply` extractor methods provided by Mapper to find the user object and post object for their respective IDs and return a boxed tuple of these or an empty box if the input was wrong. The next part takes the tuple of user and post and creates `List[String]` of their unique IDs.

For displaying the user's post we need another `star.html` file in folder `/examples/users/star/posts` to reflect the menu path we have just told `AUserPost` to find the page.

There's more...

Now this approach is different from what other frameworks do. The usual, well-known way is to rewrite the URL, either with the web server's help (`.htaccess` files) or through the framework. Lift supports that approach, too. However, we believe using menu parameters will keep your code base cleaner and easier to understand. You will not go fishing for "Where does that come from?" until you finally find who redirected to a different place, but the logic that extracts the URL parameters is kept close to the snippets that use them.

It's important that these menu locations are plugged into Boot's SiteMap, otherwise you will get an error trying to access a page that is not whitelisted. In Boot, you can use the `Hidden` LocParam to hide the defined menu entry so it never appears in any menu, but is just there, so access is granted. Also, make sure to attach the correct permissions to the entry. You can do this using the `loggedIn` parameter or another more refined parameter as you do with the other menu entries.

These parameters that Lift extracts for you from the URL are converted into their respective instances (users and posts) and passed into the snippet that needs them:

```
class AUserPost(p: (User, UserPost))
```

So all the work, from extracting the parameter from the URL to retrieving the corresponding object from the database, is done by Lift. We believe this makes the code much cleaner.

The URL we defined for user's posts (`/examples/users/1/posts/1`) is completely open. You could, for instance, spare the "posts" part and make it just `/examples/users/1/1`. To make this change you would move the `/examples/users/star/posts/star.html` file one level higher and remove the `posts` folder. Also, you would need to adapt the menu parameter that points to that page in `AUserPost`:

```
... / "examples" / "users" / * / * ...
```

I omitted the rest of the definition; it's just the same as given previously. The important piece is the missing `posts` path element to also remove this part from the URL the user would see.

Menu parameters are not the easiest part of Lift; they require some understanding of the inner processes. Don't feel discouraged when you need more than one attempt to push through. Lift provides many concepts that are far from mainstream, but that doesn't mean they are inferior ideas. On the contrary, many mainstream concepts are there because everyone uses them, not because they are the best approach.

Lift's online documentation provides some information on menu parameters at `http://simply.liftweb.net/index-Chapter-3.html#toc-Chapter-3`.

Yet a better way to learn more might be to study existing code like this example or ask specific questions on the Lift mailing list at `groups.google.com/group/liftweb`.

Lift's MegaProtoUser (Medium)

Hardly any web application can live without any kind of user management. Even a simple blog needs an admin area where you can log in to add or edit content.

To get you started quickly, Lift provides a readymade framework for managing users. In this recipe you will learn how to use it and adapt it to your needs.

Getting ready

For this recipe we will look at the User class in `code.model`. It extends Lift's `MegaProtoUser` for the user management of the example application. In that sense, let's eat our own dog food.

If your application needs to handle users, you need to build user management. That way you can provide features exactly the way you need them. That is a feasible approach, yet plan for a fair amount of time to invest that you could otherwise put into creating application-specific features.

The other approach is to use Lift's build in the `ProtoUser` and `MegaProtoUser` classes. While `ProtoUser` provides a relatively barebone user implementation, `MegaProtoUser` extends it with e-mail validation, forgot-password form, account edit form, and some other goodies you need in your user management. `MegaProtoUser` implementations exist for Mapper (SQL databases), Record (NoSQL such as MongoDB), and LDAP. If you would like to use some other persistence layer, for instance JPA, you would need to build your own user management or adapt `MegaProtoUser` for your use case.

How to do it...

To build upon Lift's `MegaProtoUser`, create the Scala artifact, `User.scala` in the `code.model` package:

```
/**
 * The singleton that has methods for accessing the database
 */
object User extends User with MetaMegaProtoUser[User] {
  override def dbTableName = "users"
  override def screenWrap =
    Full(<lift:surround with="default" at="content">
        <lift:bind /></lift:surround>)
  // define the order fields will appear in forms and output
  override def fieldOrder = List(id, firstName,
    lastName, email, locale, timezone, password, textArea)
  // comment this line out to require email validations
  override def skipEmailValidation = true

  /*
   * Add our own LocGroup to the user menu items.
   */
  override def globalUserLocParams = LocGroup("main") ::
              super.globalUserLocParams
}

class User extends MegaProtoUser[User] {
  def getSingleton = User
  // define an additional field for a personal essay
  object textArea extends MappedTextarea(this, 2048) {
    override def textareaRows  = 10
    override def textareaCols = 50
    override def displayName = "Personal Essay"
  }
}
```

How it works...

In Mapper (see this example application) and Record's case we create a `User` class that holds one user per instance and a companion object, `User`, which provides utility functions, for example querying the database.

You create your very own user management simply by extending the `MegaProtoUser` and `MetaMegaProtoUser` traits.

Let's first look at the `User` class implementation.

The bare minimum we need is to implement `getSingleton` and let this method return the `User` companion object.

That's all there is for a minimal version. We extend `MegaProtoUser`, which is parameterized with the type of the extending class (`User`).

In the remaining part of the class definition we add a sample field to `User`, a text area.

The only thing really necessary is the highlighted line, which adds a text field named `textArea` to the `User` class. The Schemifier in Boot takes care of automatically adapting the schema in order to support that field.

The other overrides you see are there for display purposes when you use the Mapper's `toForm` method to automatically generate HTML markup from the database fields. This feature is used in `LiftScreen` and `Wizard`; for example, visit the following link:

`http://simply.liftweb.net/index-Chapter-4.html#toc-Chapter-4`

Now let's look at the companion `User` object defined at the top of the preceding example code.

The `User` object is a standard Scala object that extends `MetaMegaProtoUser`.

We then override `dbTableName` to tell it the name of the database table we want our user data written to.

In the `screenWrap` section we basically define some markup that is put around a **User** form, for example around the **Create Account** form. The markup here uses the same `default.html` template that is used for all other pages. You can, however, use any template that you see fit. Make sure to wrap its definition in a box, though.

Overriding `fieldOrder` gives you a way of defining the order of fields as they will appear in generated forms. Fields not listed here are omitted.

Next we tell `MegaProtoUser` to skip e-mail validation, otherwise we wouldn't be able to create usable accounts without an attached e-mail service to send validation e-mails from. In production, however, this is a very neatly built-in feature that saves you from implementing user e-mail validation yourself.

Skipping e-mail validation makes you use the application on the spot. If e-mail validation is enabled, you will be sent a validation link that you have to click in order to access the site. To use that link you have to set up a mailer so that Lift knows how to send these e-mails. The wiki (`http://www.assembla.com/spaces/liftweb/wiki/Mailer`) gives some information on setting up a mailer and extending it with own functions.

Because we want to show the user menu items in a different menu than the menu links to the example pages, we override `globalUserLocParams` and give it our own LocParam to let the menu builder know where it should attach the menu items.

There's more...

The companion `User` object gives you very convenient access to the database for doing things, such as `User.find(id)` to find a user by his unique ID, or `User.find(By(User.email, emailAddress))` to find a user by e-mail address. The result is a boxed value that is either "Full" (found something) or "Empty" (eh, nothing there).

Of course, your queries can get more complex:

```
User.findAll(By_>(User.creationDate, date))
```

This is for selecting all users created after a certain date.

If you are using a relational database with Lift, you probably want to learn a lot more about Mapper, for instance through the following wiki:

`http://www.assembla.com/spaces/liftweb/wiki/Mapper`

We have just looked at the `MegaProtoUser` implementation for Mapper. In the *Lift and MongoDB (Advanced)* recipe we will use a slightly different incarnation to work with MongoDB.

There are many more places that you can adapt to your specific needs. We encourage you to look into the source of `MetaMegaProtoUser` and `MetaProtoUser` specifically, as these provide definitions for account edit functionality, password lost form, login hooks, and so on. If you need something different, you should copy that specific function from the Lift source into your user implementation and change it appropriately.

What do you do when the provided `MegaProtoUser` isn't enough anymore? While the suggested way of starting your application is indeed using `MegaProtoUser`, you might run into situations where you are not able to easily extend it anymore by overriding existing functionality. One way to proceed is to copy the `MegaProtoUser` source code from the Lift project into your code base and adapt it the way you want. The Lift license perfectly allows this, and it's actually a suggested way towards a richer user model.

Having said that, for most of our projects we still use `MegaProtoUser` and just extend it where it makes sense. One area might be to change the forms inside `MegaProtoUser` from simple request forms to Ajax forms, or to change the layout of the form. These things can easily be accomplished by extending the existing `MegaProtoUser`, without a need to replace it.

Handling forms (Simple)

For each application comes the time where it needs to gather input from its users. There is a conversation going on between your application, or the things your application does, and the people using it.

In this recipe we will explain form support within Lift and how you can take advantage of it.

Getting ready

As always, it's a good idea to follow this text directly in the code and observe the result in the running application. The form examples come with very fancy names, "Form 1" to "Form 3".

To create a Lift form you need the frontend markup on one side and the backend snippet on the other side. The markup is standard HTML with some Lift spice mixed in, and then the magic happens in the snippet.

How to do it...

Let's create the form markup (`form3.html` in the example application):

```
<form data-lift="NiceLiftForm?form=post">
  <p>
    <label for="animal">Animal:</label><br>
    <input id="animal" name="animal">
  </p><p>
    <label for="legs">Legs:</label><br>
    <input id="legs" name="legs">
  </p>
  <button type="submit">Submit</button>
</form>
```

Now we need to create a snippet class in the `code.snippet` package. Our example lives in `FormExamples.scala` at the location `code/task11/snippet/`; the package name of the class is still `code.snippet` though.

```
class NiceLiftForm extends StatefulSnippet  {
  private var animal = ""
  private var legs = 0

  def dispatch = {case "render" => render}
  def render = {
    def process() {
      if (legs < 2) {
        S.error("Less then 2 legs, are you serious?")
      } else {
        S.notice("Animal: "+animal)
        S.notice("Has Legs: "+legs)
        S.redirectTo("/")
      }
    }
```

```
        "@animal" #> SHtml.text(animal, animal = _) &
        "@legs" #>   SHtml.text(legs.toString, s =>
                     asInt(s).foreach(legs = _)) &
        "type=submit" #> SHtml.onSubmitUnit(process)
      }
    }
```

That's all we need to do in order to present a form to the user and process the data on the backend.

How it works...

The form we just created looks like a standard HTML form, except the `form` tag is different. In this definition you tell Lift which snippet we want to use for processing the form. In addition, the `?form=post` parameter tells Lift that it should generate the binding for handling a form. For Lift forms you do not need to specify an `action` path as this is handled by Lift. The rest of the markup is a standard form, defining labels, input fields, and a button that will submit the form to the server using the `POST` method.

Now let's look at the snippet that handles this form. This `snippet` class will bind the `form` parameters to the `instance` variables; it will preserve state when values are wrong and you need to send the form back to let the user modify the input, and it does that all in a secure manner so that no IDs or other backend-specific data gets exposed to the client.

We define private class VARs for saving the name of the animal and the amount of legs. Because our snippet is a class and not an object, this is safe to do and does not expose state to other threads.

Because we inherit from `StatefulSnippet`, which in turn inherits from `DispatchSnippet`, we need to implement a `dispatch` method to map the name of the called method to an existing method. If no name is given in the markup, just the class name, then the `render` method is assumed.

At the bottom of the `render` method we bind the form fields to the local variables and the submit button to an `onSubmit` handler. So when the button is clicked, the `process` method is invoked to actually do something with the data. We just check the amount of legs, and if too small, we return an error message (`S.error`), otherwise we return an info message (`S.notice`).

Now let's see how we ended up with that markup and snippet code.

There's more...

To compare the working form, we just created the standard HTML forms and saw how they evolve into Lift's form handling. Let's look at a standard web form and explain the Lift-specific changes from there.

In `form1.html` you will see how forms are handled in an old-fashioned way:

```
<form method="post" data-lift="UnLiftifiedForm">
  <p><label for="animal">Animal:</label><br>
    <input id="animal" name="animal"></p>
  <p><label for="legs">Legs:</label><br>
    <input id="legs" name="legs"></p>
  <button type="submit">Submit</button>
</form>
```

The only difference in a standard form is the use of the `data-lift` attribute to invoke a snippet when it's submitted:

```
def render(in: NodeSeq): NodeSeq = {
  for {
    r <- S.request if r.post_? // let's check that we have a
                                  POST request
    animal <- S.param("animal") // which animal is it?
    legs <- S.param("legs") // how many legs does it have?
  } {
    // That's the place to do something with the received
       data, we just put out a notice.
    // and redirect away from the form
    S.notice("Your Animal: "+animal)
    S.notice("Has Legs: "+legs)
    S.redirectTo("/")
  }
  // In case it's not a post request or we don't get the
     parameters we just return the html
  in
}
```

We get the request, check whether it's a `POST` request, and if it is, then we take the parameters out of the request. If they're there, we process them; if not, we just return the HTML (`NodeSeq`) we got.

This approach is cumbersome and comes with security implications. For once, the names of the `POST` parameters are fixed and known to a potential attacker, so he could easily try a replay attack. Also, if your designer changes the name of the input field in the form, you will not get the data because the names mismatch. Some other reasons for not going that way are mentioned at the following link:

`http://simply.liftweb.net/index-4.1.html#toc-Section-4.1`

Let's look at how Lift simplifies form handling. First let's look at a better way of binding form values to backend variables. The only difference in the way we define our "Liftified" form (see `form2.html`) is the following line of code:

```
<form data-lift="SimpleLiftForm?form=post">
```

We basically hand the creation of the form markup over to Lift. The rest of the form is identical with the previous example; however, the backend functionality in snippet `SimpleLiftForm` isn't:

```
object SimpleLiftForm {
  def render = {
    var animal = ""
    var legs = 0

    def process() {
      if (legs < 2) {
        S.error("Less then 2 legs, are you serious?")
      } else {
        S.notice("Animal: "+animal)
        S.notice("Has Legs: "+legs)
        S.redirectTo("/")
      }
    }

    "@animal" #> SHtml.onSubmit(animal = _) &
    "@legs" #>
            SHtml.onSubmit(s => asInt(s).foreach(legs = _)) &
    "type=submit" #> SHtml.onSubmitUnit(process)
  }
}
```

First we create two variables – `animal` and `legs` – inside the `render` method. It's important that these variables are inside the method because the surrounding object is a singleton and putting the variables there would share their content with every user accessing this form!

The real difference comes at the end of the snippet when we bind the form parameters to the method variables using CSS selectors. `"@animal"` selects the element with `name=animal`. When the form is submitted, the `SHtml.onSubmit` method is invoked and the parameter is written into the variable. For the `legs` parameter we do a conversion to `int`.

Then we bind the submit button to `SHtml.onSubmitUnit(process)`. This will invoke the `process` method when the button is clicked and the form is submitted.

That's a bit better but still not as good as it can be. For instance, when you enter a wrong value (`legs < 2` or a string), you will lose the data you entered into the `animal` field already. Now, here's a way to create Lift forms that will handle these kinds of not-so-uncommon scenarios. We use the same template as in `form2.html` with just a different snippet:

```
class NiceLiftForm extends StatefulSnippet  {
  private var animal = ""
  private var legs = 0

  def dispatch = {case "render" => render}
  def render = {
    def process() {
      if (legs < 2) {
        S.error("Less then 2 legs, are you serious?")
      } else {
        S.notice("Animal: "+animal)
        S.notice("Has Legs: "+legs)
        S.redirectTo("/")4
      }
    }

    "@animal" #> SHtml.text(animal, animal = _) &
    "@legs" #> SHtml.text(legs.toString, s =>
               asInt(s).foreach(legs = _)) &
    "type=submit" #> SHtml.onSubmitUnit(process)
  }
}
```

The first noticeable difference is that we now use a class that extends `StatefulSnippet` instead of a singleton object. `StatefulSnippet` stores state within the class itself; that's why it cannot be an object.

Because `StatefulSnippet` extends `DispatchSnippet`, we also need a `dispatch` method that maps the name of the called method to a snippet method. As you see, if no name is called in the markup, it defaults to `render`.

We have taken out the form variables from inside the `render` function into the class scope. The rest of the snippet is the same as for the other examples.

Lift provides other alternatives for saving state of forms. One way is RequestVars. You could create a snippet object and store the state of the form parameters in RequestVars. There's an example of doing just that at `http://simply.liftweb.net/index-4.4.html`.

Be aware that RequestVars keep their information for one HTTP request and subsequent Ajax requests. If you intend to access information in a following request cycle, you need to set their values again or, for instance, use `SessionVar` for storage.

No approach is better than the other. It might be that the specific problem you're dealing with would lend itself to one of the alternatives; other than that it's your personal choice which way you choose.

Lift offers more in this very important area than what we could cover in these few pages.

One notable thing is LiftScreen. **LiftScreen** is a programmatic way of creating a form from backend model classes. If your application requires a lot of input forms and if all of them follow a certain pattern, then you can create templates for the different types of forms you have and build them programmatically with LiftScreens, instead of implementing every form by hand. LiftScreen also handles input data validation, and with Lift 2.5 LiftScreen received a large overhaul that might very well be worth a look.

The other thing, building on top of LiftScreen, is Wizard. Wizards are multipage screens, complete with back and forward buttons and browser back button support. Use them if you want to gather user input in a multistep process.

More information about LiftScreen and Wizard can be found at `http://www.assembla.com/spaces/liftweb/wiki/LiftScreen`.

Form validation (Simple)

Building on our experience from the previous recipe, we want to discuss how we can validate input and present any error messages to the user.

Getting ready

You have your form ready now, ready to be filled with user input.

And because we are good developers we need to make our application, and specifically the handling of user input, robust, because some users will try to break your application by consciously entering invalid data.

The example markup for the validated form is in `validated-form.html`, and the snippet lives in `ValidatedForm.scala`.

How to do it...

Let's start by adding the `Msg` snippet to our form to provide a way to output Lift messages on the page:

```
<p>
  <label for="legs">Legs:</label><br>
  <input id="legs" name="legs"> 
```

```
      <span data-lift="msg?id=legs_msg;
            errorClass=errorMsg"></span>
</p>
```

Now you need to add validation to your snippet in order to output any error messages. We provided an example in the `ValidatedForm` snippet:

```
class ValidatedForm extends StatefulSnippet  {
  private var animal = ""
  private var legs = "0"

  def dispatch = {case _ => render}
  def render = {
    def process() {
      asInt(legs) match {
        case Full(l) if l < 2 =>
          S.error("legs_msg",
                  "Less then 2 legs, are you serious?")
        case Full(l) =>
          S.notice("Animal: "+animal)
          S.notice("Has Legs: "+legs)
          S.redirectTo("/")
        case _ =>
          S.error("legs_msg",
                  "The value you typed is not a number.")
      }
    }

    "@animal" #> SHtml.text(animal, animal = _) &
    "@legs" #> SHtml.text(legs, legs = _) &
    "type=submit" #> SHtml.onSubmitUnit(process)
  }
}
```

How it works...

In order to display messages on a page, Lift provides two built-in snippets, `Msgs` and `Msg`.

The `Msgs` snippet is used to collect messages at one place and output them in a central location, for instance at the top of the page.

The `Msg` snippet, on the other hand, is used to output specific messages targeted on a specific field.

In the case of this example we define a `<span>` element that will be filled with an error message. If no error occurs, the span is invisible. With the `id` parameter we give it a unique error ID. This is the ID that must be found within the created error in order to be displayed at this place. Using the `errorClass` parameter we tell the snippet to use `errorMsg` as the CSS class to attach to the message.

The other side of the coin is the form field validation inside the snippet.

During the validation in `process` we do a bit more elaborate parsing of the `legs` input field. We differentiate between input that is not a number and a number that is too small. In both error cases we create a message with `S.error()` giving it an ID and the message itself. Make sure the ID you assign here is the same that you used in the template. All messages created by the snippet are accumulated and will be sent back to the page when the processing finishes.

There's more...

Let's look at the message snippets in a bit more detail.

We already use the `Msgs` snippet in the `default.html` template. This snippet is responsible for showing any notification, warning, or error message that is not targeted at a specific ID. Using a snippet parameter we can tell `Msgs` to output all messages, even those with a message ID:

```
<div data-lift="Msgs?showAll=true"></div>
```

This will output all messages. The default CSS classes for these messages are `lift__noticesContainer___notice`, `lift__noticesContainer___warn`, and `lift__noticesContainer___error`, but you can tell Lift which classes to use:

```
<div data-lift="Msgs">
  <lift:error_class>errorMsg</lift:error_class>
  <lift:warning_class>warningMsg</lift:warning_class>
  <lift:notice_class>noticeMsg</lift:notice_class>
</div>
```

Additionally, the preceding example omits `showAll=true`, which causes Lift to output only those messages that are not otherwise bound on the page.

Let's look at the markup for specific messages:

```
<span data-lift="msg?id=legs_msg;errorClass=errorMsg"></span>
```

`Msg` takes the ID of the message that should be displayed and, optionally, a CSS class for styling. To set the CSS you can use the `noticeClass`, `warningClass`, or `errorClass` parameters accordingly.

The other part is the backend that creates the messages. In the simplest case, your snippet creates `S.error(<msg_id>, <error_message>)`, as seen in the `ValidatedForm` snippet.

Lift's message handling is also integrated into models managed by Mapper or Record. Together with LiftScreen (see the preceding recipe) they make for a great couple to create forms complete with validation and message display in a programmatic way. The `Mapper` or `Record` validation methods return `FieldError` instances that basically contain a `I` message and the message text.

The following Lift wiki provides further information on the `Msg/Msgs` snippets as well as other built-in functionality:

`http://www.assembla.com/spaces/liftweb/wiki/Templates_and_Binding`

While it's true that the message snippets are a part of Lift itself, it doesn't mean you cannot change them. If you need additional or different functionalities, have a look at the snippet code in the Lift code base, copy it, and adapt it to your needs.

Using Ajax (Simple)

Building responsive applications will, no doubt, lead you into the arms of Ajax. **Ajax** (**Asynchronous JavaScript and XML**—`http://en.wikipedia.org/wiki/Ajax_(programming)`) is the technology that enabled Web 2.0, and it sure will add considerable value to your applications.

Getting ready

In many other web frameworks you might have come across the logic of an Ajax call starting on the client. You would use jQuery, for instance, to create `XMLHttpRequest`, call some URL on the server, and feed the server's result into a callback and process it.

With Lift, the Ajax call is also initiated on the client side of course, but it appears different. In Lift you use CSS selectors to bind Ajax functionality to frontend elements. Once you get this difference sorted out in your head, it's really easy to understand.

So let's dive right in.

How to do it...

Let's start with a very minimal example; see `ajax.html` in the example application:

```
<span id="ajax_example1"
 data-lift="AjaxExamples.example1">Click Me</span>
```

That's all; there is markup-wise code to generate Ajax calls. It's not different to other template examples we've seen previously. Now let's look at the snippet's `render` method (`code/task13/snippet/AjaxExamples.scala`):

```
def example1 = {
  "#ajax_example1" #> ajaxButton("Click Me",
                      ()=>Alert("You clicked me"),
                      "class" -> "positive")
}
```

That's all. You just made use of Ajax in your application.

How it works...

Here we have used the already-known CSS selectors to identify template elements and do something with them (the right side). In this case we bind an Ajax button to it. This will replace the `span` element from the template with a `button` element. You can optionally pass attributes that will become part of the button markup. In this case the `positive` class is attached to `button`, giving it the friendly green color. Then when the button is pressed, the given function is evaluated and the resulting JavaScript is sent to the browser and executed, resulting in an alert box shown to the user.

Please also note the little spinner image at the top of the page when you click on the button. Lift automatically enables it for the duration of the Ajax call. You can tune its behavior in Boot.

Now let's take a closer look at the `button` example.

The button takes some text or `NodeSeq` (HTML markup) to show up inside the element and a function that is executed when the button is pressed. See, Lift takes care of the plumbing. It will insert the `button` markup into the page that's being created and add an `onclick` handler to the button. That `onclick` handler is given a unique function name that's sent back to the server. The server knows that this unique name is associated with the function attached to the `ajaxButton` method and runs it.

The result of the function is of the `JsCmd` type, which is basically a wrapped JavaScript; the server takes care of sending the result back to the client browser, and it gets executed there.

For our simple example, the resulting JavaScript is as follows:

```
alert("You clicked me");
```

But it can be of any complexity. The same mechanism would apply for opening a jQuery dialog window through some Ajax command, or sending back HTML part and replacing some part of the existing page with it.

The `Alert` command is really only a wrapper around JavaScript to help you stay in a type-safe world. You can achieve the same result with the following command:

```
"#ajax_example1" #> ajaxButton("Click Me",
                       ()=>Run("alert('You clicked me')"))
```

Now when the button is clicked, the contents of `Run` are sent to the browser and interpreted as valid JavaScript.

There's more...

The `Run` class takes any string. Lift will send the string back to the client, and there it will be executed as a JavaScript command. Using raw strings will tell you only at runtime if you made a mistake typing it; for that reason resort to the existing JavaScript commands or create your own, which is tremendously easy as well. Let's create a `SafeRun` command that will wrap the JavaScript into a try/catch block:

```
case class SafeRun(text: String) extends JsCmd {
  def toJsCmd = "try {"+text+"} catch(e) {console.log(e);}"
}
```

That's all there is to your own JavaScript command. Please do not use `console.log` like this in production; the object might not be available in the browser, and you might end up producing new exceptions.

We can now use our own command as if it was built in:

```
"#ajax_example1" #> ajaxButton("Click Me",
                     ()=>SafeRun("alert('You clicked me')"))
```

Let's look at a more complex example:

```
<form data-lift="form.ajax">
  <div data-lift="AjaxExamples.example2">
    <p>
      <label for="name">Your Name:</label><br>
      <input id="name" name="name">
    </p><p>
      <label for="city">Your City:</label><br>
      <input id="city" name="city">
    </p>
    <button type="submit">Submit</button>
  </div>
</form>
```

We define a form much like the forms from the previous recipe. This time, however, we append `ajax` to the form definition. That's the whole difference; Lift will now treat this as an Ajax form. Let's look at the snippet:

```
def example2 = {
  var name = ""
  var city = ""

  def process(): JsCmd = {
    val id = nextFuncName
    val result = "%s lives in %s".format(name, city)
    val html = <li id={id} style="display:none">{result}</li>
    AppendHtml("ajax_example2_list", html) &
    FadeIn(id, 0 second, 1 second)
  }

  "@name" #> ajaxText(name, s => {name = s;
             SetHtml("ajax_example2_name", Text(name))}) &
  "@city" #> (text(city, city = _) ++ hidden(process))
}
```

We bind two form fields, `name` and `city`. The `name` field is bound as an `ajaxText` field, which means that on every `onBlur` event the input field triggers, the value of this field is sent back to the server. The server can also return something to the browser in the form of a JsCmd. In our example we simply display the value just received in the element with the `ajax_example2_name` ID. The `city` field is a standard text field that will only be submitted when the user presses the submit button. Please note `hidden(process)`, which we attached to the last field. This generates a hidden input field in the form, and whenever this field is submitted to the server, the `process` method is executed. The reason we do not bind the submit button itself is that buttons are not serialized in an Ajax form submission, so Lift would not execute anything that is bound to the button. Binding to the hidden field gets us around this problem.

Now let's look at the `process` method. This would be the place to do some data validation and make your method return a result depending on the correctness of the received data. Perform the following steps:

1. Create a unique ID with `nextFuncName`.
2. Build our result string.
3. Build `NodeSeq` with the ID attached and `display:none`.
4. Append this `NodeSeq` to the content of the element with the `ajax_example2_list` ID (it's still hidden).
5. Then fade in the new element.

In this example we also see how you can compose a JavaScript result from smaller pieces with the `&` method. In the browser the combined JavaScript will be executed in the order it was put together.

Lift's Ajax integration is easy to use and flexible. Because of that flexibility it might take a while to get your head around it.

The framework provides many more readymade commands that you can use or adapt on your own. The commands we used in this example can be found in the `net.liftweb.http.js.jquery.JqJsCmds` object. There's also `net.liftweb.http.js.JsCmds`, which provides a bit lower-level API to JavaScript functionality.

The *Simply Lift* online book also presents an Ajax example worth looking at, at the following link:

`http://simply.liftweb.net/index-4.8.html`

Going real time with Comet (Advanced)

There were responsive applications with a desktop domain for a long time. Then along came Ajax that made the creation of web applications possible that could respond to user interactions in other ways than just reloading the whole page.

Still the other direction was missing—pushing content from the server to the browser without the user specifically requesting it. This kind of dynamic interaction is necessary for application domains such as multiplayer games, sports betting applications, and user interaction; basically anything where the user's browser should reflect changes that happened without the user's interaction.

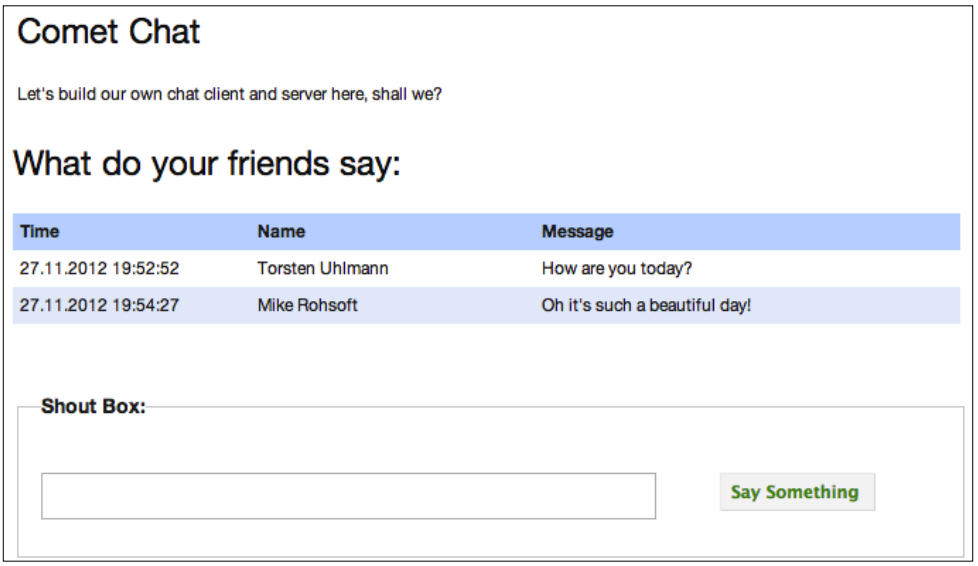
One of Lift's strength is a deeply integrated and easy-to-use facility called Comet or Server Push, which exactly enabled this kind of interaction.

Getting ready

In this recipe we will create a small, working chat client and a server that allows multiple users to interact together. Chat is that one app (`http://seventhings.liftweb.net/comet`) that is probably shown on every Lift talk. A reason might be the relatively familiar problem domain (everyone has used a chat before) and the insanely small amount of code that is needed to really pull this off.

Because it is so easy, we will build on top of the standard Lift Chat example a bit to make it even more fun to use. To follow along please use the **Comet Chat** page from the example application and open it in a couple of different browsers. This creates multiple sessions and allows you to chat with yourself, effectively seeing the messages typed in one browser popping up in all the others.

For brevity we will only show the important pieces of the markup and the code here, and we encourage you to look through the code to see the complete picture:



How to do it...

We will perform the following steps to work with Comet:

1. First we integrate the comet component on our page (comet.html):

```
<span data-lift="comet?type=ChatBrowserComponent"></span>
```

2. Let's look at the frontend component that feeds the server with the messages:

```
<input id="inp_chat" data-lift="ChatFrontend" class="title">
```

3. Embedded in our frontend form in `comet.html` is an input line that the `render` method in `code.snippet.ChatFrontend` binds to, as follows:

```
def render = {
  val userName =
      User.currentUser.map(_.shortName).openOr("Mister X")
  "#inp_chat" #> onSubmit(s => {
    if (s.trim.nonEmpty) {
      ChatServer ! Message(new Date(), userName, s.trim)
    }
    SetValById("inp_chat", "") // clear the input box
  })
}
```

4. Here we take input from the previously defined text input element, and if something was typed, we shove it over to the chat server for distribution. Let's look at the final piece of the puzzle, `code.comet.ChatServer`:

```
/**
 * Per message we create a case class
 */
case class Message(time: Date, user: String, msg: String)
object ChatServer extends LiftActor with ListenerManager { private
var msgs = Vector[Message]()
  def createUpdate = msgs

  /**
   * This method is called when the server received
   * a message.
   * We check it's of the right type, append it to our
   * saved list of messages and then send only the new
   * message to the listeners.
   */
  override def lowPriority = {
    case m: Message => msgs :+= m; updateListeners(m)
  }
}
```

5. Lastly, let's define `ChatBrowserComponent`:

```
class ChatBrowserComponent extends CometActor with CometListener {
  private var msgs: Vector[Message] = Vector()
  // register this component
  def registerWith = ChatServer

  // listen for messages
  override def lowPriority = {
    case m: Message => msgs :+= m;
         partialUpdate(appendMessage(m) & ScrollToBottom())
    case all: Vector[Message] =>  msgs = all; reRender()
  }

  def appendMessage(m: Message): JsCmd =
      AppendHtml("chat-msg-tbody", buildMessageHtml(m))

  def render = {
    Script(SetHtml("chat-msg-tbody",
           msgs.flatMap(buildMessageHtml)) & ScrollToBottom())
  }
}
```

Again, we're omitting a few helper functions here; for the complete picture, please see `code.comet.ChatBrowserComponent`.

These are all the pieces to create a fully functioning chat server.

How it works...

Let's walk through the code pieces and learn what they're doing.

Lift's Comet support is deeply integrated with the Ajax mechanisms it provides. Lift takes care of the browser-to-server connections depending on the server technology you use. You don't have to do any plumbing to set it up properly and securely; all you need to do is to integrate the Comet component in your page as you see in the code snippet under step 1 in the *How to do it...* section of this recipe.

The `comet.html` page also includes a small form to let you type in a message and also a container to show all the chat messages that have been typed by you and other users. Embedded in our form is the input line shown in step 2.

We use the snippet code under step 3 to bind to that line and get our hands at the input text. Because the surrounding form in `comet.html` is an Ajax form, there will be no full page reload; instead, the form data is serialized and pushed to the server. We take the content of the input box in the `onSubmit` method, validate it, and if the text input is not empty, we send it to the chat server.

The highlighted line of code under step 3 creates a `Message` object with a date, the name of the current user, and the received message, and sends it over to the actor object, `ChatServer`. Actors are a great way to have your application process tasks independent of each other, while one actor is not blocking another one. Actors are like mailboxes. You can send them work, like in our case using the `!` method and letting the actor process it without blocking the rest of your application. The `!` method is like a fire-and-forget method defined in the LiftActor library. It takes the message, puts it into the ChatServer's mailbox, and immediately continues the execution of this thread without waiting for the actor. The actor in turn will process the messages in its mailbox one by one in a separate thread.

The `ChatServer` instance, as shown under step 4, stores a list of all received messages in `msgs`. New listeners that register with the server will receive this list (that's what we say with the `createUpdate` method) and so can push a list of all messages to the browser, not just the latest one.

Finally, in the `lowPriority` method we define `PartialFunction` that will check on received items from the mailbox against our specification—we expect to receive `Message` instances. If we receive something else, we simply ignore it. If we do receive a new message, we append it to our internal `msgs` list and then call `updateListeners` with the latest received message. That will send this received message to each listener. A **listener** is an actor itself and thus does not block the server. The listener actor also has a mailbox, just like the server actor, and will process it sequentially.

To complete this picture let's look at `ChatBrowserComponent` shown under step 5, which is the final piece of the puzzle. This is the Comet component that will receive messages from a central Comet server instance and push them to the page.

First we register the listener actor with the server so that we will receive updates.

The central piece of the listener actor is also its `lowPriority` method, which is called for items in its mailbox in the order they arrive:

```
// listen for messages
override def lowPriority = {
  case m: Message => msgs :+= m;
       partialUpdate(appendMessage(m) & ScrollToBottom())
  case all: Vector[Message] =>  msgs = all; reRender()
}
```

In the first case we receive only one message. That is the usual case. Exactly when the server calls `updateListeners(m: Message)`, this one message is put into the listener's inbox and then processed by the `lowPriority` method.

So the first case block will match; we add the message to our internal list, create HTML code from it, and construct a JavaScript command to append the row to the message table. We create our own JsCmd case class called `ScrollToBottom`. We could have used a method that emits a `Run` command here, but this example shows you how easy it is to create your own JavaScript encapsulations.

The second match block is to catch the `createUpdate` calls of the server when the listener is instantiated. In that case we take the list of messages we receive and save it ourselves, and then just call the `reRender` function, which will render all of the message items.

We also keep our own list of messages, because after a page refresh the render method of the listener actor is called. In that case we take the list of items that we received previously, render all of them, and push them to the browser in one batch.

It is worth noting that although this is a very basic example, it still makes use of Lift's built-in security. Lift templates are of the `NodeSeq` type (basically well-formatted XML); only at the last moment, when the page is served to the client browser, it is transformed into a character sequence. Lift takes care of proper encoding and escaping characters that could lead to vulnerabilities.

There's more...

While the chat server is working, it is by no means complete. You could extend it with groups or channels, for instance. A user can create a channel, and others can join a channel or request an invite. The list of channels could itself be published through Comet push. And maybe you want to save chat messages in the database for later reference. Keeping the messages all in memory is fast, but not very robust when you need to restart the server.

Lift allows you to add more than one Comet actor to any given web page. If you give a name to your Comet actor, you can update it independently, like you could have several chat rooms on one page. Lift will take care not to use too many open connections, which could lead to starvation issues, but will multiplex them through a small number of channels.

This recipe also touched up actors, one way to achieve concurrency without worrying too much about the details such as locking and serialization. If you are not familiar with the concept of actors, it's a great opportunity to dig into it, for instance using the introduction at `http://www.scala-lang.org/node/242`.

Read on into the next recipes to learn about using NoSQL databases with Lift.

Lift and MongoDB (Advanced)

For a few years now, NoSQL databases are on the advance in many problem domains. MongoDB is one of these representatives. We won't advocate MongoDB over any other available database solution. There are many use cases, and while MongoDB fits perfectly in some, it might not be the best fit for others; having said that, we have worked extensively with MongoDB and find it a great fit into the Lift-powered landscapes.

Getting ready

To work with us through this and the next recipe, you need to have MongoDB installed somewhere on your network. Head over to `http://www.mongodb.org/display/DOCS/Quickstart` and find the very-easy-to-follow installation instructions for your system. You don't need to create a database instance or schema; Mongo does all that for you.

For this and the next recipe, please use the other (`lift_howto_9786_mongo_tpl`) project, which specifically uses MongoDB.

How to do it...

We need to tell Lift to connect to a running Mongo instance at startup. We can use a simple configuration:

```
MongoDB.defineDb(DefaultMongoIdentifier, new Mongo, "packt")
```

This connects to a Mongo instance running locally with no user or password using a database called `packt`. Or we could use a more production-ready configuration, as follows:

```
val defaultDbAddress = Props.get("mongo.default.url")
  .map(url => new DBAddress(url)).openOr(new DBAddress(
      Props.get("mongo.default.host", "localhost"),
      Props.getInt("mongo.default.port", 27017),
      Props.get("mongo.default.name", "packt")
```

```
  ))

(Props.get("mongo.default.user"),
 Props.get("mongo.default.pwd")) match {
   case (Full(user), Full(pwd)) =>
    MongoDB.defineDbAuth(DefaultMongoIdentifier,
            new Mongo(defaultDbAddress),
            defaultDbAddress.getDBName, user, pwd)
   case _ =>
    MongoDB.defineDb(DefaultMongoIdentifier,
    new Mongo(defaultDbAddress), defaultDbAddress.getDBName)
}
```

How it works...

The first thing working with a database technology of any kind is to integrate it somehow into your Lift environment. That means when Lift starts up, it should create a connection to the database and release it when it shuts down. The place to configure that is Lift's Boot class. The easy way of connecting to Mongo is shown in the first example, which just connects to a Mongo instance running locally without any username or password using the `packt` database.

The second way of connecting to Mongo covers a number of options. First we create `defaultDbAddress` of the `DBAddress` type either by shoving in the configuration value from a PROPS file (`127.0.0.1:27017/packt`, for instance) or by reading the address parameters separately and falling back to defaults if they are not available.

Next we check if a username and password are set in the relevant PROPS file. If they are, we instantiate a MongoDB connection using a username and password, or without them if they are missing.

Lift includes two different database abstraction layers, Mapper and Record. Mapper targets SQL databases, while Record seems to be more used along with alternative, NoSQL databases such as MongoDB. Please note that you are not tied to using a specific database abstraction, not even the ones that Lift provided. You could use JPA, Hibernate, or roll your own. Lift gives you many choices here to go along with the technology that best fits your project requirements. Having said that, using the provided abstractions sure comes with benefits. Mapper and Record integrate well with LiftScreen for example, and are generally tuned to work within the Lift environment. We use both in production and will continue to do so.

To make the application work we need to create a user model based on Record. We used a slightly modified version of `MegaProtoUser`, which we simply include in the project, along with the other sources. Next we need to create a user model. You find the code in `code.model.User` in the Mongo example application. It looks a lot like the Mapper version we've worked with until now, and for good reason: they all share the common `ProtoUser` trait.

There's more...

MongoDB is based on the **JavaScript Object Notation** (**JSON**) object format, from the way it saves data according to the way you query the database.

A standard query of the user collection by e-mail address would look like the following line of code:

```
db.packt.users.find({email: "tuhlmann@guessmyaddress.de"})
```

The Scala code using the LiftJson package to beautify the code reads like the following:

```
val list: List[User] = User.findAll(("email" -> email))
```

The problem with that notation (SQL has the same problem) is that it's not type safe. If my database field is called `email_adr` instead of `email`, the code would compile and the query would execute, but it would always return empty. So it would be desirable to get some more type checking in here. Hold on, in the next recipe we will get there!

We used an adapted version of Record's `MegaProtoUser` to show the MongoDB integration. Lift's current implementation of that class is a little broken and is currently being worked on. There's a good chance Lift 2.5 will come with a fully working one. Please note that this only affects the Record incarnation of `MegaProtoUser`; the Mapper one is totally fine.

Another option for integrating MongoDB into Lift is using Tim Nelson's Mongo-Auth module at `https://github.com/eltimn/lift-mongoauth`. It goes a bit of a different way: it factors out the different screens for registrations, login, and so on. So the user implementation itself becomes smaller. The module at `https://github.com/eltimn/lift-mongo.g8/` provides a templates application that uses auth-mongo together with Twitter Bootstrap and provides a really nice starting ground for your own application.

Sometimes it might be desirable to use SQL and NoSQL databases side by side. Maybe there is transaction-heavy stuff you wish to process in a relational database; or you need to connect to existing data on one side but want to store new or additional data in a NoSQL database benefitting from their easier-to-use, less-rigid structure.

Lift allows you to use multiple databases and also multiple mapping mechanisms side by side. So you could use Mapper for accessing relational databases and Record for NoSQL access. While this approach might come with benefits for your project depending on your use case, it comes with the additional cost of bridging different databases and architectures. Depending on how tightly coupled objects from one database are with their counterparts in the other (database) world, it might be a small or big task to get the two connected. There's no rule for all here. You really need to check your project requirements. Trying to get along with one database, though, is a way to avoid unnecessary work.

The next tasks show how we can use an add-on library specifically designed for Mongo and Lift to create type-safe queries in MongoDB.

MongoDB and Rogue (Advanced)

In this recipe we will show you how you can write type-safe and easy-to-read database queries with Rogue.

The folks at `foursquare.com` also use Lift for their hugely popular service. And they also use Mongo. Now we can only imagine that after a couple hundred queries and countless spelling errors in the search fields they decided to write up an easy-to-use Scala DSL that would help them avoid these bumps in the future. They did, and they kindly open sourced their solution, Foursquare Rogue, at the following link:

`https://github.com/foursquare/rogue`

We will walk you through the installation, first usage steps, and a couple of things you can do with it.

Getting ready

This recipe, same as the previous recipe, is documented in the Mongo-based example application. And as with the previous recipe, in order to run the application, you need to have a MongoDB server accessible in your network. It doesn't matter if Mongo runs on your developer machine or remotely. Installation is easy and the process of connecting to it is described in the previous recipe.

To use Rogue within your application you need to load its library. In SBT-based Lift applications this is done in `build.sbt`:

```
libraryDependencies ++= Seq(
  ...
  "com.foursquare" %% "rogue" % "1.1.8" intransitive()
)
```

That's all. And that's actually the same pattern you use for integrating any other library, which is available through open-source Maven repositories. The next time SBT starts it will download the library and add it to the build path.

You may have spotted the keyword `intransitive` in the dependency declaration. You use it to exclude Rogue's dependencies from being included as your own application dependencies. Rogue is usable with different versions of Lift. In order to bind the correct Lift version you want to use, you add it explicitly to your build configuration and tell SBT not to bother about the dependencies that come along with Rogue.

For our examples we've built a very simple data model. For once we have the `User` object. So go ahead and create a bunch of users. Since we do not send out validation e-mails, the e-mail addresses you enter don't have to match—you just need to remember them.

For each created user we create a random number of the `Note` objects that we assign to the user. That is done automatically in the background and will give us something to select upon. Both the `User` and `Note` models can be found in the `code.model` package.

Let's dive right in. The **Rogue Examples** page of the example application also contains the queries demonstrated here executed on your MongoDB. Feel free to experiment with them.

How to do it...

Perform the following steps:

1. Let's play a bit with the database. Select all users and map their `User` objects to table rows and display them on the page. The snippet for that is `RogueExamples.allUsers`, and the database query can be simply expressed, as follows:

```
def forAllUsers3: NodeSeq = (User fetch).flatMap(mapToRow)
```

2. Select all users with a `.com` domain in their e-mail addresses, as follows:

```
def findAllComEmail(): List[(String, String)] = {
  val pattern = """.*\.com""".r
  User where (_.email matches pattern)
  select(_.firstName, _.email) fetch()
```

3. Count all the `Note` objects that were created in the last 24 hours:

```
def countNotesInLast24h =
    Note where (_.date after 24.hours.ago) count
```

4. Display the `Note` objects attached to one user. The following is the snippet:

```
def notesByUser = {
  "@email-address" #> ajaxText("", e => {
    if (e.trim.nonEmpty) {
      val html = findNotesOfUser(e).flatMap(mapToRow)
      SetHtml("rogue-ex3-body", html)
    } else Noop
  })
}
```

5. Define a helper function as follows:

```
def findNotesOfUser(email: String): List[Note] = {
  val user: Box[User] =
      (User where (_.email eqs email) get())
  user.map(u => (Note where (_.userId eqs u.id)
                      fetch())).openOr(Nil)
}
```

You can see these examples in action in the `rogue.html` page.

How it works...

Rogue provides a DSL that makes it very natural and straightforward to query the database. In order to use it, you need to import the heap of implicit conversions that Rogue comes with:

```
import com.foursquare.rogue.Rogue._
```

After that its complete functionality is at your disposal.

Rogue queries appear very simple and readable.

Let's take the code under step 1 from the *How to do it...* section of this recipe as an example. We fetch all users (because we didn't enter any query); this would return `List[User]`. Then we map that list into `NodeSeq` (a table row) and convert `List[NodeSeq]` into `NodeSeq` by "flattening" it. `flatten` and `map` can be expressed together with `flatMap`.

The code under step 2 shows how we would select all users with a `.com` domain in their e-mail addresses. This query also shows how to select only specific values from your model object with the `select (...)` operator.

Selecting case-insensitive or parts of a string is a little tricky. Mongo does not support operators such as `%` or `LIKE` that you may be familiar with from relational databases. But it supports patterns matching with regular expressions. So we create a pattern that matches all e-mail addresses with a `.com` domain at the end and then use Rogue's `matches` operator for selection.

Counting records is also simple, as step 3 displays. We use Lift's TimeHelpers to create the current date minus 24 hours. Using Rogue's `count` operator will return the number of found elements instead of the elements themselves. `count` reduces the data exchanged between your application and the database, so it's a preferred way of counting the number of elements compared to selecting them all into a list and then counting the size of that.

As a final example in step 4, let's display the notes attached to one user. We provided a small Ajax form in `rogue.html`, where you can enter an existing e-mail address. The user for that e-mail address will be selected along with his notes, and these will then be pushed back to the page by Ajax. The snippet refers to a helper function, `findNotesOfUser` (shown in step 5), where the real work of selecting these notes happens.

First we find the `User` object with the accompanied e-mail address. Rogue's `get()` operator returns `Box[User]`. If none was found, the box is "Empty", otherwise it contains a user. Then, if the user was found, we select all the `Note` objects with the user's `userId` set and return that list, or Nil (the empty list) if no user is found.

There's more...

We only showed you a few simple queries, just enough to whet your appetite. You can also modify data and use Mongo's `findAndModify`, an atomic operation that let's you find something and modify it before any other process can change the data. It's an equivalent to "select for update" in the SQL world and immensely useful if you work with multithreaded applications; a web application, for instance.

There is not tremendously much information about Rogue. Be sure to read the blog posts at `http://engineering.foursquare.com/2011/01/21/rogue-a-type-safe-scala-dsl-for-querying-mongodb/` and `http://engineering.foursquare.com/2011/01/31/going-rogue-part-2-phantom-types/`.

We found the best way to learn what's possible with the library is to look at the extensive test cases that come with it at the following link:

`https://github.com/foursquare/rogue/blob/master/src/test/scala/com/foursquare/rogue/QueryTest.scala`

We find Rogue is a great DSL that fits right into Lift's record, and it makes using MongoDB in Lift so much nicer. Of course it doesn't provide any new functionality per se; it's a type-safe wrapper on top of the Scala driver for Mongo. Everything that you can do with Rogue can be done with the plain Mongo driver. However, the type-safe and easy-to-read queries is a huge feature in itself, and we're just happy we don't have to miss that.

Building a REST API (Medium)

Imagine you run into a situation where you want to open a part of your application to third-party apps, or you want to build native applications with access to the same functionality you use based on the Web. These are reasons to start thinking about implementing an API that can be accessed from other applications. **REpresentational State Transfer** (**REST**) is a great choice for building your API that will most likely fit your requirements. REST is a de facto standard for building web-based APIs and as it happens Lift comes with great built-in support for that protocol.

Getting ready

We can't go much into the details of REST. If you're not familiar with it, please use your favorite search engine or book seller. The basics of REST are what's already there in the HTTP protocol. HTTP provides the `GET`, `POST`, `PUT`, and `DELETE` request methods, which differentiate the type of action that is accomplished on the server. A `GET` method retrieves something from the server, `PUT` might add an item, while `POST` modifies an existing item. There's much more to it, yet here we want to show how you can build a REST API with Lift's simplifying helper classes.

The example for this task is contained in the SQL example application. Check out the menu item appropriately named **REST Example**. We will use the `User/UserPost` data model that we created for the *Building dynamic menu structure (Advanced)* recipe. We will create a simple API to play with these items and will access them from the aforementioned web page; we won't make you install a mobile application just to try it out.

We will create a REST API that will manipulate the list of users we have in our system. The API will receive and return JSON data, as this is very easy to provide on the server side and also to parse on the client side. Another option would be XML that we omit here. The API is also only accessible for logged-in users—we are using Lift's session support to check that. Another option would be to use the stateless API and use OAuth for authentication of your requests.

How to do it...

First let's create an object that will handle the API calls, which we create in the `code.lib` package (`code/task17/lib/UserRest.scala`):

```
object UserRest extends RestHelper {
  case class UserData(id: Long, name: String, email: String)
  serve {
    case "api" :: "user" :: "list" :: Nil Get _
        if User.loggedIn_? => anyToJValue(listAllUsers)
    case "api" :: "user" :: "list" :: AsLong(id) :: Nil Get _
        if User.loggedIn_? => anyToJValue(listUser(id))
  }

  def listAllUsers(): List[UserData] = {
    User.findAll.map{user => UserData(user.id.is,
                     user.shortName, user.email.is)}
  }

  def listUser(id: Long): Box[UserData] = {
    for (user <- User.find(id)) yield {
      UserData(user.id.is, user.shortName, user.email.is)
    }
  }
}
```

That object will hold the code to process and respond to API calls. Next we hook this object up with the Lift processing system so that Lift can include it in request processing. We do this in the Boot class, as follows:

```
LiftRules.dispatch.append(UserRest)
```

We add this line to the `boot` method to let Lift know about our API methods. If it wouldn't know about them, calls to these URIs would simply be ignored or rejected.

How it works...

In the code snippet given in the preceding section, we added a function that will return all available user IDs together with a bunch of other data and one that will return one user, provided a user ID is given.

We add a call to the `serve` method (multiple calls are possible within one object), which expects partial functions of the `PartialFunction[Req, () => Box[LiftResponse]]` type, a function that takes a request and creates a function that when called will yield `Box` of `LiftResponse`. In our examples we try not to use too much implicit magic—there are some places that can be simplified even more, but on the other hand it makes it harder to understand what's going on.

Let's dissect this partial function.

The following two extractor calls are identical:

- `case "api" :: "user" :: "list" :: Nil Get req =>`
- `case Get("api" :: "user" :: "list" :: Nil, req) =>`

The first argument to the extractor is the URI path of the request in the form of a list. The second argument is the request itself. If you are not interested in the `req` instance, you can ignore it by writing "_" instead.

That's the left-hand side of the partial function, using one of the extractors provided by `RestHelper`. You could, for instance, also use `JsonGet` or `XmlGet` if you want to distinguish between two types.

On the right-hand side we need to provide something that can be converted into a LiftResponse. Lift provides several implicit conversions that help us on the way. Our partial function outputs something of the `JValue` type, for instance, which is then implicitly converted into `()=>Box[JsonResponse]`. The `anyToJValue` method that we call on the list of our user case classes is a helper method of `RestHelper` that uses LiftJson's `Extraction.decompose` method to create JSON from case classes.

Right now everyone can call this API function without any kind of authentication. Let's lock it down so it can only be called by a user who authenticated himself:

```
case "api" :: "user" :: "list" :: Nil Get _
     if User.loggedIn_? => anyToJValue(listAllUsers)
```

The addition of `User.loggedIn_?` checks if there is a session around with an authenticated user in it. If you use stateless REST without any session, you can use OAuth to authenticate users.

It's equally simple to extract a value from the URI path, and for instance, select a user according to the given ID:

```
case "api" :: "user" :: "list" :: AsLong(id) :: Nil Get _
         if User.loggedIn_? => anyToJValue(listUser(id))
```

We extract `id` from the path and use it to look up a user record. We wrap it with an `AsLong()` extractor to make sure we only accept the `id` values of the `Long` type. `listUser` returns `Box` of `UserData`, filled if the user was found or empty if not. `RestHelper` converts the result into the appropriate response. Have a look at the **REST Example** page and test a few IDs.

There's more...

One way to simplify this even more would be to create a companion object `UserData` that would come with an `unapply` method. Scala uses these `unapply` methods to take in one or more values (such as a unique ID) and create a class from the data. The `UserData` object would take the unique ID and query the database itself, returning `Option[UserData]` that we could then, also implicitly, convert into a JSON object. The partial function could then look something like the following code snippet:

```
case "api" :: "user" :: "list" :: UserData(user) :: Nil Get _
         if User.loggedIn_? => user: JValue
```

Lift offers you a notation of the `serve` method that allows you to write a common path prefix only once, keeping you from repetition:

```
serve ("api" / "user" prefix {
  case "list" :: Nil Get _ if User.loggedIn_? =>
       anyToJValue(listAllUsers)
  case "list" :: AsLong(id) :: Nil Get _ if User.loggedIn_? =>
       anyToJValue(listUser(id))
})
```

After the `serve` method you write the path that is common for all partial functions in that block. This notation would effectively serve the same URI path as the preceding one, it just keeps you from repeating the prefix path or makes it easier to change the prefix path.

Please also keep in mind that you can have multiple `serve` methods in one `api` object. Of course you can also create multiple `api` objects and register them in Boot, which might be a feasible strategy to keep your API code modular.

We can only scratch the surface of possible things, though we hope that you have seen how easy it is to provide a REST API to your application and customers, all fully integrated into Lift.

To continue this journey, David Pollak's online book *Simply Lift* contains a thorough introduction to REST within Lift at the following link, which is very worth reading:

`http://simply.liftweb.net/index-Chapter-5.html`

This tutorial also explains how you would offer `Post`, `Put`, or `Delete` functionalities in your REST API, which is of course important if you want to create, modify, or delete data. `RestHelper` offers a lot more. If you are looking for some particular functionality or just want to learn what else it has to offer, feel free and encouraged to fetch the source from GitHub at `https://github.com/tuhlmann/packt-lift-howto` and take a look. If you have set up the Eclipse IDE, you can use `sbteclipse` to create the project files. Call it like this in the sbt shell in order to fetch and link the Lift sources to your project automatically:

```
eclipse with-source=true
```

The following Lift wiki also holds some examples for you to explore:

`http://www.assembla.com/spaces/liftweb/wiki/REST_Web_Services`

Integrating Twitter Bootstrap (Medium)

After weeks of sweat and long hours you spent in intimate relationship with your mouse and keyboard, you have created a great application that you can truly be proud of. Now the only thing that needs work is the look and feel of your work to make its presentation a truly outstanding experience.

As developers, the nitty-gritty details of user interface design are usually not our strongest quality, and it might be wise to accept help in this area. In the same way you build your backend code on the mature foundation of Lift, you should choose the right framework for the frontend presentation in the browser.

In this recipe we will walk you through installing and using Mongo-Auth, a Lift module that combines the Twitter Bootstrap framework with user and role management based on MongoDB. The example application presented here is heavily based on the Lift Mongo template application at `https://github.com/eltimn/lift-mongo.g8`.

Getting ready

For this recipe we create a new example project, `lift_howto_9786_bootstrap_tpl`. This example application can be used the same way as our other examples with the same sbt commands, the only difference is a new module that we added to its dependencies and the new functionality that we can leverage. Also please note that the application is based on MongoDB instead of SQL.

How to do it...

Perform the following steps to integrating Twitter Bootstrap:

1. To start with Mongo-Auth you need to add a dependency line to sbt's build script `build.sbt`:

```
{
  val liftVersion = "2.4"
  libraryDependencies ++= Seq(
    "net.liftweb" %% "lift-mongodb-record" % liftVersion,
    "net.liftmodules" %% "mongoauth" % (liftVersion+"-0.3"),
    "ch.qos.logback" % "logback-classic" % "1.0.0",
    "org.scalatest" %% "scalatest" % "1.6.1" % "test",
    "org.eclipse.jetty" % "jetty-webapp" % "7.6.0.v20120127" %
                          "container"
  )
}
```

   Version `0.3` was the latest version available at the time of this writing.

2. After starting sbt again, the new dependency should be fetched from the repository and be available in the project.
3. Now run the example application with `container:start` and enjoy a readymade template application.
4. Now check and adapt MongoDB settings in `src/main/resources/props/default.props`:

```
mongo.default.host = 127.0.0.1
mongo.default.port = 27017
mongo.default.name = lift-bootstrap
```

5. Check your Mailer settings in the same file:

```
mail.charset = UTF-8
mail.smtp.port = 25
mail.smtp.auth = true
mail.smtp.host=smtp.example.com
mail.smtp.user=email_user_name
mail.smtp.pass=password
```

This would produce the following screenshot:

How it works...

The application is fully working. You can sign in and edit your profile. It even fetches your Gravatar images and shows it when you're logged in. And try to resize your browser. Make it as small as a mobile screen and see how the content on the page reflows to give the best possible experience on different screen sizes.

Let's walk through the different settings and learn how you can adapt the example to your own needs.

Mongo-Auth tries to extract most configuration from Boot into distinct objects. These objects are located in the `code.config` package. `MongoConfig` holds the connection data to the Mongo database; this is very similar to the configuration that we used in the Mongo and Rogue examples, just in a different file.

You can adapt the `MongoConfig` code directly, but actually adjusting a few property values as shown previously should be all that's necessary. Even if these are missing and you have a local MongoDB running, the application should be able to start and fallback to sensible defaults.

To configure the application you need to use the correct PROPS file; `default.props` is the one for development, and `production.default.props` is the default production file.

`code.config.SmtpMailer` holds the configuration data for connecting your application to an SMTP server used to send password reset e-mails, for instance.

The preceding property values show how you can set up a connection to a sending SMTP server by adjusting the property values.

There's more...

The most interesting configuration, though, the SiteMap, is found in `code.config.Site`.

The configuration of the menu structure is not quite as straightforward as in our example applications. The reason is simply that the menu used in the Bootstrap template is a tad more complex and flexible.

Two LocGroups have been defined to arrange the menu items accordingly:

```
object MenuGroups {
  val SettingsGroup = LocGroup("settings")
  val TopBarGroup = LocGroup("topbar")
}
```

We defined these LocGroups in a way that we can also access them from other places and don't have to repeat string values.

We added a LocParam named `LocIcon` that we use to add icon classes to the different menu items:

```
case class LocIcon(cssIconClass: String*) extends AnyLocParam
```

And we use it like this:

```
val home = MenuLoc(Menu.i("Home") / "index" >>
    TopBarGroup >> LocIcon("icon-home", "icon-white"))
```

We add icon classes (see the Bootstrap manual for available icon types) to these menu items and extract them later in `MenuSnips.scala`.

The biggest difference compared to the other template applications is the `User` model. Mongo-Auth does not use `MegaProtoUser` but its own implementation contained in Mongo-Auth. This implementation does not include the registration and edit forms as `MegaProtoUser` does; it rather has externalized them in `snippet.UserScreen` using the LiftScreen foundation, which lets you build forms programmatically rather than by repetitive definition in markup. This approach is different from `MegaProtoUser`, but it keeps the `User` model smaller and easier to extend.

It should be mentioned that the build file adds some tools necessary to build a production-ready application.

For once the Bootstrap CSS framework comes as LESS source files and needs to be compiled into CSS. The sbt build uses the `less-sbt` plugin to compile LESS files and append them into one CSS file. This way you can develop your styles modular but don't pay a penalty for having multiple small CSS files. The LESS plugin looks for files ending with `styles.less` and compiles them into files with the same name ending with `.css`. The compiled files are found in the `resource_managed` directory and included from there.

For the JavaScript files in your project, the template uses Google's Closure Compiler to minify code and merge everything in one file. The `src/main/javascript` directory contains a `script.jsm` file that acts as kind of a manifest file, it names all files to be included and also defines their order. You can create more than one JSM file; the Closure Compiler will compile all of them and save the created artifacts in the `resource_managed` directory.

Oh, and one more thing. This template comes with built-in extended session support. The **Log In** screen offers an option to stay logged in, even if you leave the site or the server gets restarted. That's one thing less you have to take care of to implement for your users.

We are using the Mongo template in our own application with great success. It's a great basis that lets you get started quickly and build your application on a solid foundation at the same time.

There might come a time, however, when you would like to change things in the Mongo-Auth module itself. Of course, the best way to do that is to contribute to the official project and thus help to keep it going. However, nothing would stop you to merge the code into your own application and change it there directly, maybe because you need to implement some unique changes.

The Mongo-Auth module can be found on GitHub at `https://github.com/eltimn/lift-mongoauth`, and you can find the g8 (Giter8) template application at `https://github.com/eltimn/lift-mongo.g8`.

To work with Bootstrap you should definitely study their GitHub page at `http://twitter.github.com/bootstrap/`. Bootstrap is a framework containing many different parts, both CSS styles and JavaScript components. It's worth spending some time with it to know your way around the grid, the form components, buttons, and all that other neat stuff.

Thank you for buying Instant Lift Web Applications How-to

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING

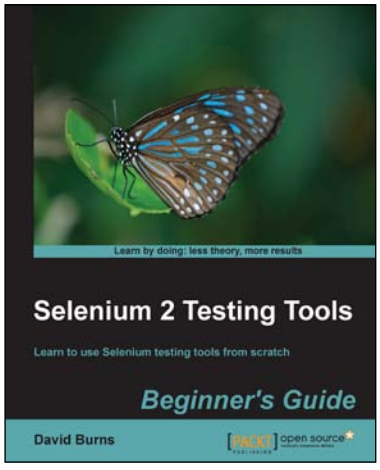
Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0 Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

---



Selenium 2 Testing Tools: Beginner's Guide

ISBN: 978-1-84951-830-7 Paperback: 232 pages

Learn to use Selenium testing tools from scratch

1. Automate web browsers with Selenium WebDriver to test web applications
2. Set up Java Environment for using Selenium WebDriver
3. Learn good design patterns for testing web applications