

Timothy Perrett

Lift IN ACTION

The simply functional
web framework for Scala



MEAP

 MANNING



**MEAP Edition
Manning Early Access Program
Lift in Action production version**

Copyright 2011 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=660>

Table of Contents

Part 1: Getting started

1. Introducing Lift
2. Hello Lift

Part 2: Application tutorial

3. The Travel auction application
4. Customers, auctions, and bidding
5. Shopping basket and checkout

Part 3: Lift in detail

6. Common tasks with Lift Webkit
7. SiteMap and access control
8. HTTP in Lift
9. AJAX, wiring, and Comet
10. Persistence with Mapper
11. Persistence with Record

12. Localization

13. Distributed messaging and Java enterprise integration

14. Application testing

15. Deployment and scaling

Appendixes

A. Introduction to Scala

B. Configuring an IDE

C. Options and Boxes

Part 1

Getting started

The first two chapters of this book introduce the Lift framework and demonstrate how you can get everything set up and ready for your first development.

Chapter 1 starts by introducing both Scala and Lift concepts, complete with high-level explanations and samples. The aim is to give you a grounding in what is a fundamentally different way of thinking. In chapter 2, you'll be building upon the basis laid down in chapter 1 by constructing your very first Hello World application, which will involve the most basic Lift steps. In these chapters, you'll see first-hand how Lift leverages a *view-first architecture* and how easy it is to get up and running with the Lift web framework.

1

Introducing Lift

This chapter covers

- An overview of Scala and Lift
- Lift's history and design rationale
- An overview of Lift's structure

Lift is an exciting new way of building web applications that feature rich, highly interactive user experiences. Lift is built atop the powerful functional programming language Scala, which lends itself to writing exceedingly concise but powerful code. By leveraging Scala, Lift aims to be expressive and elegant whilst stressing the importance of maintainability, scalability, and performance.

The first section of this chapter will introduce Scala and functional programming, including some examples of how the language compares and contrasts to the more familiar style of imperative programming. The second section introduces Lift and discusses how it differs from other web programming tools available today. Lift is largely a conceptual departure from many of the traditional approaches to building web frameworks; specifically, Lift doesn't have a controller-dispatched view system and opts for an idea called *view first*. This chapter discusses these core design goals and introduces these new ideas at a high level. Throughout the course of the book, the concepts outlined here will be expanded on in much greater detail, and you'll see concrete examples to assist you in getting up to speed.

If you're reading this book but are new to Scala programming, you can find a rough guide in appendix A that will show you the ropes and give you a foundation for making use of Lift. If you want to get serious with Scala, I highly recommend looking at the other Scala titles published by Manning: *Scala in Action* by Nilanjan Raychaudhuri, and then the more advanced *Scala in Depth* by Joshua Suereth.

1.1 What is Scala?

Scala (<http://www.scala-lang.org/>) is a powerful, hybrid programming language that incorporates many different concepts into an elegant fusion of language features and core libraries. Before delving any deeper, let's just consider how functional programming differs from imperative programming with languages such as Java and Ruby, and what a functional programming language actually is.

As the name implies, functional programming languages have a single basic idea at their root: functions. Small units of code are self-contained *functions* that take type A as an argument and return type B as a result; this is expressed more directly in Scala notation: $A \Rightarrow B$. How this result is achieved is an implementation detail for the most part; as long as the function yields a value of type B, all is well.

NOTE Functional programming languages often derive from a mathematical concept called lambda calculus. You can read more about it on Wikipedia: http://en.wikipedia.org/wiki/Lambda_calculus.

With this single concept in mind, it's possible to boil down complex problems into these much smaller functions, which can then be *composed* to tackle the larger problem at hand; the result of function one is fed into function two and so on, ad infinitum. The upshot of such a language design is that once you wrap your head around this base level of abstraction, many of the language features can be thought of as higher levels built upon this foundation of basic functions.

Immutability is another trait that marks out functional languages against their imperative cousins. Specifically, within functional languages the majority of data structures are *immutable*. That is to say, once they're created there is no changing that instance; rather, you make a copy of that instance and alter your copy, leaving the original unaltered.

Martin Odersky, however, wanted to fuse object orientation and functional programming together in one unified language that could compile and run on the Java Virtual Machine (JVM). From here, Scala was born, and consequently Scala compiles down to Java bytecode, which means that it can run seamlessly on the JVM and interoperate with all your existing Java code, completely toll free. In practical terms, this means that your existing investment in Java isn't lost; simply call that code directly from your Scala functions and vice versa.

With this fusion of programming styles, Scala gives you the ability to write code that's typically two or three times more concise than the comparative Java code. At the same time, the Scala code is generally less error-prone due to the heavy use of immutable data constructs, and it's also more type-safe than Java, thanks to Scala's very sophisticated type system.

These are the general concepts that make up functional programming, and upon which Scala is built. To further exemplify these differences, table 1.1 presents some examples that illustrate the differences in Scala's approach compared to imperative code. If you don't know

Java, don't worry: the examples here are pretty easy to follow, and the syntax should be fairly readable for anyone familiar with Ruby, PHP, or similar languages.

Table 1.1 Comparing Java and Scala styles of coding

| Java | Scala |
|---|---|
| <p>When building class definitions, it's common to have to build so-called getter and setter methods in order to set the values of that instance. This typically creates a lot of noise in the implementation (as seen in the Java example that follows). Scala combats this by using the <code>case</code> modifier to automatically provision standard functionality into the class definition. Given an instance of the <code>Person</code> case class, calling <code>person.name</code> would return the name value.</p> | |
| <pre>public class Person { private int _age; private String _name; public Person(String n, int a){ _age = a; _name = n; } String name(){ return _name; } int age(){ return _age; } }</pre> | <pre>case class Person(name: String, age: Int)</pre> |
| <p>Most applications at some point have to deal with collections. The examples that follow create an initial list and then produce a new list instance that has the same animal names, but in lowercase. The Java example on the left creates a list of strings, then creates a second empty list, which then has its contents mutated by looping through the first list and calling <code>toLowerCase()</code> on each element. The Scala version achieves the exact same result by defining a function that should be executed on each element of the list. The Scala version is a lot more concise and does the exact same thing without the code noise.</p> | |
| <pre>List<String> in = Arrays.asList("Dog", "Cat", "Fish"); List<String> out = new ArrayList<String>(); for(String i : in){ out.add(i.toLowerCase()); }</pre> | <pre>List("Dog", "Cat", "Fish") .map(_.toLowerCase)</pre> |

These are just some of the ways in which Scala is a powerful and concise language. With this broad introduction to Scala and functional programming out of the way, let's learn about Lift.

1.2 *What is Lift?*

First and foremost, Lift (<http://liftweb.net/>) is a sophisticated web framework for building rich, vibrant applications. Secondly, Lift is a set of well-maintained Scala libraries that are used by many other projects within the broader Scala ecosystem. For example, the Dispatch HTTP project (<http://dispatch.databinder.net/Dispatch.html>) uses Lift's JSON-handling library extensively for parsing JSON within the context of standalone HTTP clients. This book, however, really focuses on Lift as a web framework, and it's here that our story begins.

User behavior online in recent years has changed; people now spend more time than ever online, and this means they want the way they interact with online services to be more intuitive and natural. But building such rich applications has proven to be tough for many developers, and this often results in interfaces and infrastructures that aren't really up to the job or user expectations. Lift aims to make building real-time, highly interactive, and massively scalable applications easier than it has ever been by supporting advanced features like Comet, which allow you to push data to the browser when its needed, without the user having to make any kind of request for it.

In fact, Lift has been designed from the ground up to support these kinds of systems better than anything else. Building interactive applications should be fun, accessible, and simple for developers. Lift removes a lot of the burdens that other frameworks place on developers by mixing together the best ideas in the marketplace today and adding some unique features to give it a component set and resume that are unlike any other framework you have likely come across before. Lift brings a lot of new ideas to the web framework space, and to quote one of the Lift community members, "it is not merely an incremental improvement over the status quo; it redefines the state of the art" (Michael Galpin, Developer, eBay). This departure from traditional thinking shouldn't worry you too much, though, because Lift does adopt tried and tested, well-known concepts, such as convention over configuration, to provide sensible defaults for all aspects of your application while still giving you a very granular mechanism for altering that behavior as your project dictates.

One of the areas in which Lift is radically different is in how it dispatches content for a given request. Unlike other frameworks, such as Rails, Django, Struts, and others, Lift doesn't use the traditional implementation of Model-View-Controller (MVC), where view dispatching is decided by the controller. Rather, Lift uses an approach called "view first." This is one of the key working concepts within Lift, and it affects nearly everything most developers are used to when working with a framework. Specifically, it forces you to separate the concerns of content generation from content rendering markup.

In the early days of web development, it was commonplace to intermingle the code that did business computations with the code that generated the HTML markup for the user interface. This can be an exceedingly painful long-term strategy, as it makes maintaining the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=660>

code problematic and tends to lead to a lot of duplication within any given project. Conceptually, this is where the MVC pattern should shine, but most implementations still give the developer the ability to write real code within the presentation layer to generate dynamic markup; this can add accidental complexity to a project when the developer unwittingly adds an element of business or process logic to the presentation layer. It takes programmers who are very disciplined to ensure that none of the business or application logic seeps into the view. Lift takes the standpoint that being able to write interpreted code within markup files can lead to all manner of issues, so it's outlawed completely: this ensures that your templates contain nothing but markup.

The view-first idea in Lift really inherits from the broader design goals upon which Lift was conceived. The following sections will cover these design goals, provide some details about Lift's architecture, and give you an overview of the Lift project structure and community.

1.2.1 Lift design goals

The design goals upon which Lift was based have remained fairly constant features of the project. For example, the belief that complex problems, such as security, should be the responsibility of a framework, and not of the developer, have remained central ideals. In short, Lift's design goals are security, conciseness, and performance. Let's just take a look at these in closer detail and consider how they impact you when using Lift as a general-purpose web development framework.

SECURITY

The web can be a dangerous place for developers who don't fully appreciate the potential attacks their applications could come under. There are whole rafts of malicious techniques, including cross-site request forgery (CSRF), cross-site scripting (XSS), SQL injection, and lots, lots more. Many developers can't keep up with the constantly changing world of security threats, let alone fully understand how to effectively and securely protect their applications.

To this end, Lift provides protection against common malicious attacks without the need for the developer to do any additional work or configuration: Lift just does the right thing. Whenever you make an AJAX call, use Comet, or even build a simple form, Lift is right there in the background securing the relevant processing from attack. Lift typically does this by replacing input names and URLs with opaque GUIDs that reference specific functions on the server; this essentially completely eliminates tampering, because there is no way for an attacker to know what the right GUID might be. This comprehensive security is covered in more detail in chapters 6 and 9.

A nice illustration of Lift's security credentials is the popular social media site foursquare.com, which runs on Lift. Even Rasmus Lerdorf, the inventor of PHP and infamous security pundit, was impressed by not being able to find a single security flaw!¹

¹ Tweet on Rasmus Lerdorf's Twitter stream: <http://twitter.com/rasmus/status/5929904263>.

CONCISENESS

If you have spent any time coding in a moderately verbose imperative programming language like Java, you'll be more than familiar with the value of conciseness. Moreover, studies have shown that fewer lines of code mean statistically fewer errors, and overall it's easier for the brain to comprehend the intended meaning of the code.²

Fortunately, Scala assists Lift in many aspects with the goal of conciseness; Scala has properties, multiple inheritance via *traits*, and as was touched on earlier, it has a complex type system that can infer types without explicit type annotations, which gives an overall saving in character tokens per line of code that you write. These are just some of the ways in which Scala provides a concise API for Lift, and these savings are coupled with the design of the Lift infrastructure, which aims to be short and snappy where possible, meaning less typing and more doing.

PERFORMANCE

No matter what type of application you're building for use on the web, no developer wants his or her work to be slow. Performance is something that Lift takes very seriously, and as such, Lift can be very, very quick. As an example, when using the "basic" Lift project, you can expect upward of 300 requests per second on a machine with only 1 GB of RAM and a middle-of-the-road processor. In comparison, you should see upwards of 2,000 requests per second on a powerful 64-bit machine with lots of RAM. Whatever your hardware, Lift will give you really great throughput and blistering performance.

1.2.2 View-first design

Lift takes a different approach to dispatching views; rather than going via a controller and action, which then select the view template to use based upon the action itself, Lift's view-first approach essentially does the complete opposite. It first chooses the view and then determines what dynamic content needs to be included on that page. For most people new to Lift, trying not to think in terms of controllers and actions, can be one of the most difficult parts of the transition. During the early phases of Lift development, there was a conscious choice taken to *not* implement MVC-style controller-dispatched views.

In a system where views are dispatched via a controller, you're essentially tied to having one primary call to action on that page, but with modern applications, this is generally not the case. One page may have many items of page furniture that are equally important.

Consider a typical shopping-cart application: the cart itself might feature on multiple pages in a side panel, and a given page could contain a catalog listing with the mini shopping cart on the left. Both are important, and both need to be rendered within the same request. It's at this very point that the MVC model becomes somewhat muddy, because you're essentially forced to decide which is the primary bit of page content. Although there are

² For more information, see Gilles Dubochet's paper, "Computer Code as a Medium for Human Communication: Are Programming Languages Improving," (École Polytechnique Fédérale de Lausanne, 2009). <http://infoscience.epfl.ch/record/138586/files/dubochet2009coco.pdf?version=2>.

solutions for such a situation, the concept of having a primary controller action for that request immediately becomes less “pure.”

In an effort to counter this problem, Lift opts for the view-first approach. Although it's not a pattern you may have heard about before, the three component parts are view, snippet, and model—VSM for short. This configuration is illustrated in figure 1.1.

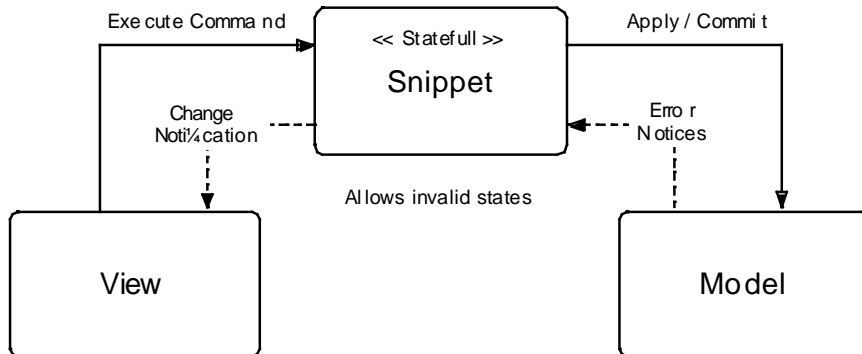


Figure 1.1 A representation of the view-first design. The view invokes the snippets, which in turn call any other component of the application business logic.

Figure 1.1 shows that the view is the initial calling component within this architecture, and this is where the “view first” name comes from. Let’s now take a moment to review each element within the view-first setup.

VIEW

Within the context of view first, the view refers primarily to the HTML content served for a page request. Within any given Lift application, you can have two types of view:

- Template views that bind dynamic content into a predefined markup template
- Generated views in which dynamic content is created, typically with Scala XML literals

Template views are the most commonly used method of generating view content, and they require that you have a well-formed XHTML or HTML5 template. It’s important to note that Lift doesn’t allow you to use view code that’s invalid; this means that when you’re working with a design team, if their templates W3C-validate, you know they’ll work with Lift because the snippet invocations are also part of the markup. This ensures that designers don’t inadvertently introduce problems by altering framework-specific code within the template, which is a common problem with other frameworks.

Generated views are far less common, but sometimes they’re used for quick prototyping by using Scala XML literals.

Whichever route you choose to take, the view is the calling component in the architecture, and as such you can invoke as many different (and completely separate) snippets as you like from within any given view. This is a core idea within Lift: views can have more than a single concrete purpose. This helps to minimize the amount of code duplication within an application and lends itself nicely to a pure model of component encapsulation.

SNIPPET

Snippets are rendering functions that take XML input from within a given page template and then transform that input based upon the logic within the snippet function. For example, when rendering a list of items, the template could contain the markup for a single item, and then the snippet function would generate the markup for an entire list of items, perhaps by querying the database and then iterating over the result set to produce the desired list of items.

There is a very tight and deliberate coupling between the snippet and the XML output. The snippet isn't intended to be a controller, such as those found in the MVC design pattern, nor is it meant to take on any control-flow responsibilities. The snippet's sole purpose within Lift is to generate dynamic content and mediate changes in the model back to the view.

MODEL

In this context, the model is an abstract notion that could represent a number of different things. But for most applications, it will represent a model of persistence or data (irrespective of the actual process it undertakes to get that data). You ask the model for value *x*, and it returns it.

In terms of Lift's view-first architecture, the snippet will usually call the model for some values. For example, the snippet might request a list of all the current products in an ecommerce application or ask the model to add an item to the user's shopping cart. Whatever the operation, when the model is asked to do something, it applies whatever business logic it needs to and then responds appropriately to the snippet. The response could include validation errors that the snippet then renders to the view.

The actual mechanism for updating the view isn't important for this discussion (full page load, AJAX, or some other method). Rather, the model responds and the response is passed to the view via the snippet.

1.2.3 Community and team

Since the very beginning, the Lift team has always been very diverse; right from the early days, the team grew in a very organic fashion and has continued to do so over recent years. Today the Lift core team consists of professional and highly talented individuals not only from all over the world but in a bewildering array of different market sectors. This gives Lift its vibrancy and overall well-rounded approach.

If you're new to the Lift community, welcome. It's a very stimulating place, and you'll find that the majority of our team members on the mailing list or hanging out in IRC will assist you if you get stuck with something. Although I hope that this book will cover most of the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=660>

things you might want to know about Lift, there will inevitably be things you wonder about as you continue to use Lift in your own projects. To that end, take a look at the resources listed in table 1.2.

Table 1.2 Helpful Lift resources that can be found online

| Resource | Description |
|-----------------|--|
| Main Lift site | http://liftweb.net |
| | First and foremost is the main Lift homepage. Here you'll find the latest news about Lift, regularly updated as time goes by. This page also has links to the source code, the issue tracker, and the wiki. |
| Assembla | https://www.assembla.com/wiki/show/liftweb |
| | Lift moved to the Assembla platform for its wiki and bug-tracking requirements some time ago, and since then it has accumulated a fair amount of community-created articles. |
| Mailing list | http://groups.google.com/group/liftweb |
| | The Google group is the official support channel for Lift. If you have a question, you can come to the mailing list and find a friendly, responsive community that will be more than happy to answer your questions. |
| IRC channel | #lift on freenode.net |
| | IRC isn't as popular as it once was, but you'll still find some of the Lift team hanging out in IRC from time to time. |

Now that you've had a brief overview of the Lift framework and its evolution, let's get into some technical details as to what it can actually do and how it can help you be more productive and produce higher quality applications.

1.3 Lift features

During the past three years, the Lift codebase has exploded in size and now features all manner of functionality, from powerful HTTP request-response control, right through to enterprise extensions like a monadic transaction API and Actor-based wrappers around AMQP and XMPP.

Lift is broken down into three top-level subprojects: Lift Core and Lift Web, Lift Persistence, and Lift Modules. We'll now take a closer look at each module to give you an overview of its structure and functionality.

1.3.1 Lift Core and Lift Web

There are two modules that make up the central framework: *Core* and *Web*. The Core consists of four projects that build to separate libraries that you can use both with and without Lift's Web module. The Web module itself builds upon the Core and supplies Lift's sophisticated components for building secure and scalable web applications. The Web module itself is made up of three projects: the base web systems and two additional projects that provide specialized helpers. Figures 1.2 and 1.3 depict the various modules and their layering.

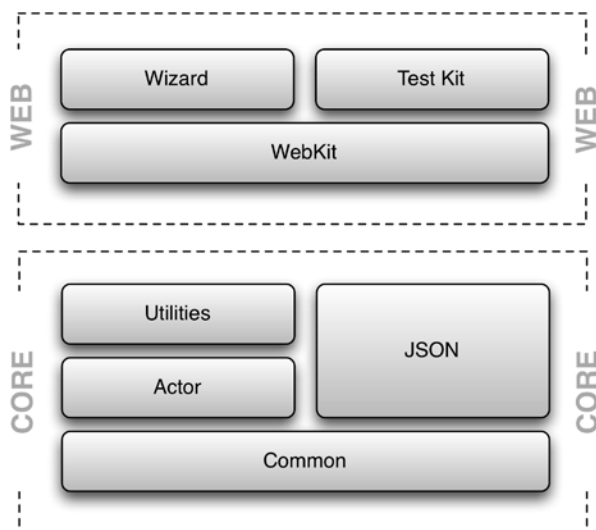


Figure 1.2 An illustration of the module dependencies within the Lift Core and Web subprojects

Let's spend some time going through each module in figure 1.2, working from the bottom up, and discuss their key features and functionality.

LIFT COMMON

The Lift Common module contains a few base classes that are common to everything else within Lift. Probably most important of all, Lift Common can be used in projects that aren't even web applications. Utilities like `Box`, `Full`, and `Empty` (discussed more in appendix C) can be exceedingly useful paradigms for application development, even if the application isn't using any other part of Lift. Lift Common also includes some base abstractions that make working with the logging facade SLF4J (<http://www.slf4j.org/>) much simpler.

LIFT ACTOR

Actors are a model for concurrent programming whereby asynchronous messaging is used in place of directly working with threads and locks. There are several actor implementations

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=660>

within the Scala ecosystem, and Lift has its own for the specific domain of web development. To that end, Lift Actor provides concrete implementations of the base actor *traits* that are found within Lift Common (more information on traits can be found in appendix A).

LIFT UTILITIES

During the development of web applications, there are invariably things that can be reused because there are common idioms in both your and other peoples' work. Lift Utilities is a collection of classes, traits, and objects that are designed to save you time or provide convenience mechanisms for dealing with common paradigms.

A good example is the handling of a time span. Consider the following code, which defines a pair of `TimeSpan` instances by way of *implicitly converting* a regular integer value into a `TimeSpanBuilder`:

```
10 seconds
1 hour
```

Simplistic helpers provide an easy-to-use dialect for handling even complex subjects like time and duration. This example shows both hour and second helpers, where both lines result in `net.liftweb.util.Helpers.TimeSpan` instances.

Here's another example from the `SecurityHelpers` trait. It hashes the string "hello world" with an MD5 algorithm:

```
md5("hello world")
```

Once again, Lift Utilities provides simple-to-use helper methods for common use cases found within web development—everything from handling time to hashing and encrypting values and much more.

LIFT JSON

Probably one of the most popular additions to the Lift Core grouping, Lift JSON provides an almost standalone package for handling JSON in a highly performant way. Needless to say, JSON is becoming one of the standards within the emerging online space, so having great support for it is quite critical for any web framework. The parser included within Lift JSON is approximately 350 times faster than the JSON parser that's included in the Scala standard library—this gives Lift blisteringly fast performance when serializing back and forth to JSON.

You might be wondering if Lift can only parse JSON quickly, or if it also provides a means to construct JSON structures. Well, Lift JSON provides a slick domain-specific language (DSL) for constructing JSON objects.

Let's take a quick look at a basic example:

```
val example = ("name" -> "joe") ~ ("age" -> 35)

compact(JsonAST.render(example))
```

This example defines a value in the first line, which represents a JSON structure with Scala tuples. This structure is then *rendered* to JSON by using the `compact` and `render` methods from the `JsonAST` object in the second line. Here's the output:

```
{"name": "joe", "age": 35}
```

As you can see, this is a straightforward `String` and `Int` construction from the DSL, but we'll cover more in-depth details of Lift-JSON in chapter 9. All you need to know for now is that Lift's JSON provisioning is fast and very versatile.

LIFT WEBKIT

Finally we get to the central part of Lift's web toolkit. The WebKit module is where Lift holds its entire pipeline, from request processing right down to localization and template rendering. For all intents and purposes, it's the main and most important part of Lift.

Rather than covering the various parts of WebKit in detail here, table 1.3 gives an extremely brief overview of some of the core components and notes the chapter that addresses it in more detail.

Table 1.3 Features of Lift WebKit

| Feature | Description | Chapter |
|--------------------------------------|---|----------------|
| Snippet processing | Snippets are the core of Lift's rendering and page-display mechanism. | 6 |
| SiteMap | SiteMap provides a declarative model for defining security and access control to page resources. | 7 |
| HTTP abstraction | Although Lift typically operates within a Java servlet container, it's totally decoupled from the underlying implementation and can run anywhere. | 8 |
| Request-response pipeline processing | The whole request and response pipeline is contained within WebKit, as are the associated configuration hooks. | 8 |
| REST components | REST features allow you to hook into the request-response pipeline early on and deliver stateless or stateful web services. | 8 |
| Secure AJAX | All the AJAX processing and function mapping infrastructure lives in WebKit. | 9 |
| Rich Comet support | The Comet support Lift provides is one of the main features WebKit offers. | 9 |

Although you aren't familiar with Lift syntax or classes just yet, here's an example of a real-time Comet clock to give you a flavor of the kinds of things contained within the WebKit project.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=660>

Listing 1.1 CometActor clock

```
import scala.xml.Text
import net.liftweb._,
  util.Schedule, util.Helpers._,
  http.CometActor, http.js.JsCmds.SetHtml

class Clock extends CometActor {

  Schedule.schedule(this, Tick, 5 seconds)           #A

  def render = "#clock_time *" replaceWith timeNow.toString

  override def lowPriority = {
    case Tick =>
      partialUpdate(SetHtml("clock_time", Text(timeNow.toString)))
      Schedule.schedule(this, Tick, 5 seconds)
  }
}
#A Schedule redraw
```

With only a few lines of code, you get a clock that pushes the updated time to the browser, so it will appear as if there's a *live* clock in the user's browser. All the complexities associated with Comet, like connection handling, long polling, and general plumbing are handled by Lift right out of the box!

1.3.2 Lift Persistence

The vast majority of applications will at some point want to save their data for later use. This typically requires some kind of backend storage, and this is where Lift Persistence comes into play. Lift provides you with a number of options for saving your data, whether it's a relational database management system (RDBMS) or one of the new NoSQL solutions.

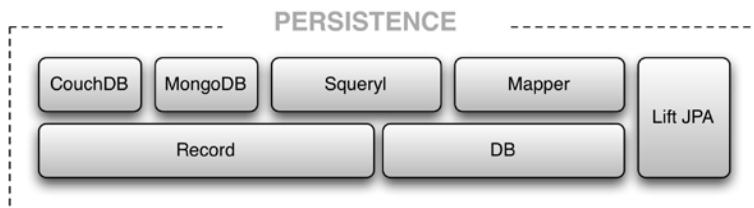


Figure 1.3 Dependency structure of persistence within Lift

There are three foundations for persistence, as depicted in figure 1.3; the following subsections take a look at these base components.

LIFT DB AND MAPPER

The vast majority of applications you'll write will no doubt want to communicate with an RDBMS of some description, be it MySQL, SQL Server, or one of the other popular storage

systems. When you're working with Lift, Mapper provides you with a unified route for persistence.

At a high level, Mapper takes a design direction that's similar, but not completely faithful to the Active Record pattern. Mapper provides you with an object-relational mapping (ORM) implementation that handles all the usual relationship tasks, such as one-to-one, one-to-many, and many-to-many, so that you don't have to write SQL join queries manually. But when you want to write that raw SQL, perhaps for performance reasons or by preference, you can easily pull back the covers and write SQL directly.

Mapper is unified into many parts of Lift and thus has several advantages out of the box over other solutions that are available within the Scala ecosystem. Consider this very basic example of the Mapper API and how it can be used:

```
User.find(By(User.email, "foo@bar.com"))

User.find(By(User.birthday, new Date("Jan 4, 1975")))
```

Notice that this code is quite readable, even without a prior familiarity with the Mapper API. For example, in the first line, you want to find a user by his or her email address. In the second line, you're finding a user by their birthday.

LIFT JPA

The Java Persistence API is well known in the wider Java space, and, being Java, it can work right out of the box from the Scala runtime, which shares the common JVM platform. Unfortunately, because JPA is Java, its idiomatic implementation gives it a lot of mutable data structures and other things that are typically not found within Scala code—so much so that you might well choose to avoid writing Java-like code when you're working with Scala.

To that end, a module was added to Lift's persistence options to wrap the JPA API and give it a more idiomatic Scala feel. This module significantly reduces the Java-style code that you need to write when working with JPA and the associated infrastructure. This is covered in more detail in chapter 13.

LIFT RECORD

This is one of the most interesting aspects of Lift Persistence. Record was designed with the idea that persistence has common idioms no matter what the actual backend implementation was doing to interact with the data. Record is a layer that gives users create, read, update, and delete (CRUD) semantics and a set of helpers for displaying form fields, operating validation, and so forth. All this without actually providing the connection to a concrete persistence implementation.

Currently, Record has three backend implementation modules as part of the framework: one for working with the NoSQL document-orientated storage system CouchDB (<http://couchdb.apache.org/>), a second for the NoSQL data store MongoDB (<http://www.mongodb.org/>), and finally a layer on top of Squeryl (<http://squeryl.org/>), the highly sophisticated functional persistence library. These implementations could not be more different in their underlying mechanics, but they share this common grounding through Record because of the abstract semantics the Record infrastructure provides.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=660>

At the time of writing, Record is still fairly new. As time goes by, more and more backend implementations will come online, and perhaps eventually the Mapper RDBMS code will also be merged with Record.

Here is a sample from the CouchDB implementation that queries a CouchDB `people_by_age` JavaScript view:

```
Person.queryView("test", "people_by_age", _.key(JInt(30)))
```

It's important to note that third-party backend implementations for Record are starting to appear in the Scala ecosystem, and although they aren't a bona fide part of Lift, they're available on github.com and similar services.

UNDERSTANDING YOUR USE CASE

As you have probably grasped from the framework overview, Lift has many different components, some of which overlap in their intended usage. This is not a legacy growing pain, quite the opposite: it's deliberate. With Lift there is often more than one way to reach an acceptable solution, and the factors that dictate which route you take are largely application-specific and depend on the particular problem domain you're working with.

Throughout the course of this book, you'll see a range of different approaches to solving problems with Lift. Often the different methods are equally as good, and which you choose is a matter of preference or style. For example, in chapter 14 you'll learn about three different approaches to dependency injection with Scala. These approaches ultimately achieve very similar results, but depending upon your team, environment, or application, one may be a better fit than the others. That's something you must experiment with for yourself to get a feel for which is going to work best for you.

The next section discusses some plugins, or modules of ancillary code that are also available as part of the Lift project. They may help you in building your applications and getting up to speed with less plumbing.

1.3.3 Lift Modules

Lift Modules is where the project houses all the extensions to the core framework. Unlike the other "groups" of subprojects within Lift, the modules are more organic and have little or no relation to one another. Each module is generally self-contained regarding the functionality it provides.

Rather than go through each module in detail here, table 1.4 lists the modules available at the time of writing.

Table 1.4 Available add-on modules supplied as part of Lift

| Module | Description |
|--|---|
| Advanced Message Queue Protocol (AMQP) | Actor-based wrapper system on AMQP messaging |
| Facebook integration | API integration module for the popular social networking site |

| | |
|--|---|
| Imaging | Selection of helper methods for manipulating images |
| Java Transaction API (JTA) integration | Functional style wrapper around the Java Transaction API |
| Lift state machine | State machine tightly integrated with WebKit and Mapper |
| OAuth | Building blocks for creating the server component of OAuth |
| OAuth Mapper | Extension to the OAuth module to use Mapper as a backend |
| Open ID | Integration module for using OpenID federated providers |
| OSGi | For those who want to run their Lift app within an OSGI container |
| PayPal | Integration module for PayPal PDT and IPN services |
| Test kit | Helpers for writing tests concerning the HTTP operations in Lift |
| Textile | Scala implementation of a Textile markup parser |
| Widgets | Selection of helpful widgets (such as calendaring, Gravatar, and JavaScript autocomplete) |
| XMPP | Actor based wrappers around XMPP message exchange |

At the time of writing, the available modules are located within a separate Git repository (<https://github.com/lift/modules>), and the community was discussing making the addition of new modules available to non-core team committers.

If you want to create your own modules, it's just a case of depending upon the parts of Lift that you wish to extend. Typically this means creating a small library of your own that depends upon WebKit and extends or implements the relevant types. To use this custom module within another application, you only have to provide some kind of initialization point that will wire the relevant materials into that Lift application during startup. That's all there is to it.

1.4 Summary

In this chapter, we've taken a look at both Scala and Lift and outlined their major conceptual differences from more "traditional" web frameworks. Lift provides developers with a very capable toolkit for building interactive, scalable, and highly performant real-time web applications. These themes really underpin the core design goals of Lift: security, conciseness, and performance.

As the implementer of a Lift application, you don't need to worry about the bulk of security issues prevalent in other systems: Lift does that for you. The framework is always there securing element names and URIs without you having to intervene. In addition to security, idiomatic Lift application code tends to be brief and make use of powerful Scala language features to create an API that's readable, maintainable, and performant.

Lift also differs quite wildly from other frameworks available today in that it doesn't implement controller-dispatched views as many MVC frameworks do. Instead, Lift implements its own view-first architecture that gives you a far purer model for creating components and modularity within your code. Your rendering logic takes the form of neat, maintainable functions rather than monolithic stacks of special classes.

Finally, the majority of the code contained within the Lift framework is either running in production, or is a distillation from live production code. To that end, you can have absolute confidence in Lift when building your enterprise applications.

Without further ado, let's move on to setting up your environment and getting your very first Lift-powered application up and running.