

The simply functional web framework for Scala

Lift

IN ACTION

Covers Lift 2.x

Timothy Perrett





Lift in Action

by Timothy Perrett

Chapter 12

brief contents

PART 1	GETTING STARTED	1
	1 ■ Introducing Lift	3
	2 ■ Hello Lift	20
PART 2	APPLICATION TUTORIAL	37
	3 ■ The auction application	39
	4 ■ Customers, auctions, and bidding	60
	5 ■ Shopping basket and checkout	83
PART 3	LIFT IN DETAIL	105
	6 ■ Common tasks with Lift WebKit	107
	7 ■ SiteMap and access control	140
	8 ■ HTTP in Lift	160
	9 ■ AJAX, wiring, and Comet	187
	10 ■ Persistence with Mapper	223
	11 ■ Persistence with Record	259

- 12 ■ Localization 282
- 13 ■ Distributed messaging and
Java enterprise integration 293
- 14 ■ Application testing 317
- 15 ■ Deployment and scaling 347

12

Localization

This chapter covers

- Lift's localization strategies
- Configuring the locale calculator
- Utilizing Java localization infrastructure

In this age of globalization and the apparent consolidation of many cultures, it may surprise you to know that there are still between 6,500 and 10,000 different languages spoken in the world today. The exact figure isn't known, but you can be sure that our world has a large amount of linguistic diversity. Traditionally, this has caused a large number of problems for software engineers because these different languages often have cultural differences that are specific to a certain locale.

Take something as simple as a greeting in two English-speaking countries, like Great Britain and Australia; the former would likely use *Good afternoon* and the latter would likely use *Good'ay*. Strictly speaking, both countries are English-speaking, but cultural differences lead to colloquialisms becoming commonplace and a subsequent divergence from the original dialect. As you might imagine, there's an infinite amount of variation in presentation styles and content, such as when comparing Latin-based text to Arabic or Hebrew, which are both written right-to-left, and not

left-to-right. All these things and more combine to make writing software for the international market subtly difficult.

Fortunately both the Java platform and Lift provide some great features to help you make the best of this rather complex situation. The first section of this chapter looks at how you can implement Lift's localization helpers in your templates and application code. Then, the second section covers how you actually obtain localized content—where your code or template defines one of Lift's localization helpers, and where the content that ultimately replaces it comes from. There are three possible options and they're all illustrated with examples in section 12.2.

First, though, let's get into how you define localized content in your templates and code.

12.1 Implementing localization

The Java platform as a whole has good support for localization and for working with global systems that have large numbers of locale-specific, colloquial idioms. Because the JVM has had a localization strategy for a long time, the design of the system is largely oriented toward building desktop applications (such was the trend at the time). Lift provides a system that builds on top of JVM's robust localization infrastructure, specifically the `Locale` and `ResourceBundle` classes supplied with the standard Java distribution, but it also adds several very useful facilities that make localizing *web applications* much more flexible.

What is a locale?

If you're new to localization, the term *locale* is one you'll see thrown around a lot. There are two aspects to localization: languages and countries. These two words alone describe orthogonal parts of a culture, but locale defines the combination of the two. For example, French is spoken both in France and in North Africa, but the cultures aren't the same. French spoken in France is assigned a locale of `fr_FR`, whereas French spoken in Morocco is assigned a locale of `fr_MA`.

Strictly speaking, you can have a `java.util.Locale` instance that only represents the language, but generally a locale is a unification of both aspects. It's only useful to represent a language as a locale if you want to group together countries that speak the same language: such as American English, British English, and so on.

The language identifiers used by the Java locale infrastructure are the ISO 639-1 language codes, and the country codes are those in the ISO 3166-1 definitions. These are international standards and are widely used in many different systems.

Lift's localization strategy can be broken down into a couple of component parts: resources and templates.

Resources define locale-specific objects. These objects could be pretty much any type you like, but the most common use case is a string that needs to be presented in

several languages. Lift will look at the resource bundles it has available, and load the appropriately localized string if it exists.

The second element of Lift's localization strategy is part of the template selection. Once you've got the localized content, there are often still issues with presenting that content. For example, German text is roughly 30 percent longer than English text for the same content, and this can have fairly serious impacts on the visual presentation of an application; ultimately, you need to find room for all this extra text. With this in mind, Lift allows you to be smart about templating. Consider a page called `index.html`; this would be fine for English, French, and Spanish, but perhaps you need to rearrange some of your markup to make space for the longer German text. In this case, you could simply have a secondary file called `index_de_DE.html`.

Before implementing the localizations, you need to instruct Lift on how it should determine which locale is the correct one to use for a request. This is done via the `localeCalculator` property of `LiftRules`.

12.1.1 *Implementing locale calculator*

Out of the box, Lift will grab the locale from the `Accept-Language` header of the incoming request. If a given request doesn't specify this particular header, Lift will assume the locale of the server to be the locale for this request. For example, if the JVM of the container you're running your application on has a locale of `en_GB`, and the request has not specified a preferred language with the `Accept-Language` header, the server will respond with the British English version.

NOTE Most browsers in use today make web requests with what are known as accept headers. Their purpose is to indicate to the server application what responses would best fit the user's client. Accept headers exist for content types and even language. The latter is specifically helpful for determining what language the user speaks. For example, if the user has their browser language set to French (`fr_FR`), you can be fairly sure they would prefer reading content in French. You can read more about the Accept Language header in RFC 2616: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>.

It's also highly likely that you'll want to provide users with some control over the locale, and you may want the application to remember the user's locale choice for the next time they visit. To illustrate this, let's assume you want to control the locale based on the query parameter `hl`. This would mean that if a request was sent to `/foo?hl=en_GB` or `/bar?hl=fr-FR`, the locale would be set to the value specified by the `hl` query parameter by parsing the string value and constructing a `java.util.Locale` instance.

All requests that hit your Lift application are routed through `Liftrules.localeCalculator`, and this is where the parsing and construction of the locale will be done, based on the information contained in the request. This means you could use query parameters, cookies, and other factors to determine what the locale should be. In your `Boot` class, implement the locale calculator as shown in the following listing.

Listing 12.1 Example localeCalculator configuration

```
import java.util.Locale
import net.liftweb.http.LiftRules
import net.liftweb.common.Box
import net.liftweb.util.Helpers.tryo

LiftRules.localeCalculator = (request: Box[HTTPRequest]) => (for {
  r <- request
  p <- tryo(r.param("hl").head.split(Array('_', '-')))
} yield p match {
  case Array(lang) => new Locale(lang)
  case Array(lang, country) => new Locale(lang, country)
}).openOr(Locale.getDefault))
```

`LiftRules.localeCalculator` takes a function from `Box[HTTPRequest] => Locale`, and in this sample a for comprehension is used to extract the boxed request value to the value `r`. Given that `r` is an `HTTPRequest`, you can call the `param` method to grab a value from the query string. Because this method could fail if the value isn't present, it's wrapped in the `tryo` control structure to capture the result as a `Box[String]`. Assuming the value of `hl` is of the format `en_GB` or `en-GB`, it's then split into its two component parts to instantiate a new `Locale`. Failing that, it will use the default locale for your JVM by calling the `Locale.getDefault` method.

You'd probably want to expand on this implementation to provide something a lot more robust that saves the value into the session, a cookie, or some other place. You'll generally want to keep track of the locale rather than calculating it on every request from scratch.

There's one important thing to be aware of with `localeCalculator` and the way in which Lift handles requests. As each page view is made up of a selection of requests, the function will, by default, execute for every request that makes up the page. If your `localeCalculator` is doing anything but the default, it's a good idea to memoize the function execution. By adding a request-scoped memoize, you can ensure that your locale calculation function is only executed once in the scope of a request; for every other call on that value, the cached result will be given rather than incurring the overhead of executing the function again.

NOTE You can find more information about memoization in the Wikipedia article: <http://en.wikipedia.org/wiki/Memoization>.

The following listing shows the implementation that makes use of memoization.

Listing 12.2 Implementing memoization with localeCalculator

```
import java.util.Locale
import net.liftweb.common.Box
import net.liftweb.util.Helpers.{tryo, randomString}
import net.liftweb.http.provider.HTTPRequest
import net.liftweb.http.{LiftRules, RequestMemoize}
```

```

object localeMemo extends RequestMemoize[Int, Locale] {
  override protected def __nameSalt = randomString(20)
}

LiftRules.localeCalculator = (request: Box[HTTPRequest]) =>
  localeMemo(request.hashCode, (for {
    r <- request
    p <- tryo(r.param("hl").head.split(Array('_', '-')))
  } yield p match {
    case Array(lang) => new Locale(lang)
    case Array(lang, country) => new Locale(lang, country)
  }).openOr(Locale.getDefault))
)

```

① Memoization object

← ② Call-by-name function

In order to apply the memoization, you need to import the `RequestMemoize` type and set up a local object that extends it with two type parameters: a key type and a value type ①. The value type is `Locale`, and for simplicity sake it uses the instance `hashCode` of the request as the key. This ensures that the locale is paired with the right request, because each request will create a new `HTTPRequest` instance. Notice that in the actual implementation of the locale calculator, the `localeMemo` object ② is applied with two parameters: the `hashCode` of the incoming request and a block that's the same code from listing 12.1. This second parameter is a call-by-name parameter, which means the `localeMemo` object only executes it if it needs to.

TIP You may be wondering if it's possible to access the calculated locale value from other places in your application code, such as a snippet. Fortunately, this is usually quite straightforward: if the location you'd like to obtain the value from is within session scope, you can just call `S.locale`.

Now that Lift can determine the correct locale for each request, let's do something useful with this new information and load some localized content from both templates and your own code.

12.1.2 Localizing templates and code

Lift has a selection of ways to interact with localized content, depending upon where and what you need to localize. Broadly speaking, these can be divided into two types: localized templates and localized code.

LOCALIZED TEMPLATES

In the section introduction, we briefly touched upon the differences between languages and how, for example, English is a left-to-right text whereas Hebrew is written right-to-left (RTL). These differences in language direction have an impact on how the site looks and functions. Consider figure 12.1, which compares a page in Hebrew and in English.

Try as you might, there will usually come a point when simply using resource bundles to swap content for given positions won't cut it. Verbose European languages may



Figure 12.1 Comparing English and Hebrew localization

cause problems if you're tight on space, or implementing RTL languages may push things to the breaking point and then require some template alterations.

It's exactly these situations that Lift's template localization is designed for. The content itself is the same, but you want to change the way it's being presented. This may involve including a custom `<head>` or perhaps making some page-specific changes to accommodate content in a better manner.

When Lift receives a request and then goes looking for the template, it will also include a set of locale-specific templates in that search as well. Suppose you have `index.html` as your content file, but you want to make a special case for German. In order to implement this, you'd have both an `index.html` and an `index_de.html` file (or one that also specifies the country: `index_de_DE.html`).

This is great for per-page corner cases in different locales, but you may be wondering about what you'd do for something more global, such as a surround. Well, the same is true. Even if your content page has a call to `surround` that defines default, you can have a `default_iw_HL.html` (where `iw_HL` is the locale code for Israeli Hebrew), and that root surround would automatically be selected by Lift allowing you to use the same content for the page, but apply a RTL styling.

Finally, if you have content in a template that you want to localize, you have two possible ways to implement the Lift Loc snippet:

```
<lift:loc locid="greeting">Greeting goes here</lift:loc>
```

or

```
<p class="lift:loc?locid=greeting">Greeting goes here</p>
```

Both of these versions of template markup indicate to Lift that the content of that node should be replaced by resolving the `locid` as a key in the resource bundle system.

LOCALIZED CODE

In order to localize your own code, you need some way of telling Lift that it must grab this particular bit of content from the resource bundle rather than taking the literal from the code, much in the same way that the `Loc` snippet does in templates. For this, Lift provides a slightly odd-looking `S.?method`.

Consider these examples:

```
import net.liftweb.http.S.{?,??}

?("mything")
?("salutation", title, lastName)
??"ajax.error"
```

These are three examples of obtaining localized resources from the defined bundles. Notice that there are two methods in use here: `?` and `??`. The `?` method is used for obtaining your own resources, and then the `??` method is used to obtain resources that come bundled as part of Lift. For the most part, you'll usually implement the `?` method, but understanding the purpose of `??` is helpful if you see it in example code or you wish to override parts of Lift's functionality.

The first example in the preceding code shows the most common use case. Resource bundles that contain localized content always define a *key* to access the localized item, so imagine you have a literal string in a resource bundle that's accessible via the key "mything"—this method will just return the value as it's defined in that resource bundle.

The second example defines a formatted resource string where the value would be something like "Dear %s%s", where the `%s` is a placeholder for the variable argument list passed to the second parameter. In this case, given `?("Dear %s %s", "Mr", "Perrett")` the result would be "Dear Mr Perrett". Such strategies can be used to localize the majority of your server-side text, such as the names of buttons or other generated content.

The other item of code that you'll likely want to localize is your sitemap location declarations. Fortunately, this is simple enough. You can just adjust the definition to use the `i` method on the `Menu` object:

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

With this definition, the sitemap builds upon the `S.?method` and uses the name of the page (Home, in this example) as the key to resolve the resource.

Because localization lookup is conducted using strings for keys, you may want to handle the case where an appropriate translation doesn't exist for a particular key. Lift supplies a hook for this, which allows you to execute any function you please in the event that a localization key isn't found in any of the available resource bundles. Add the following configuration to your application `Boot`:

```
LiftRules.localizationLookupFailureNotice = Full((key, locale) =>
  logger.warn("No %s text for %s".format(locale.getDisplayName, key)))
```

These two lines instruct Lift to log a warning when a translation key isn't present. In this instance, the function just logs an error, but it could just as easily execute any code you require.

You have now seen how to configure Lift with a dynamic locale, and you've also had an introduction to how you can implement localized resources from both templates and code. With this in mind, it's time to go through the various options available for loading localized content from resource bundles. The next three sections deal with these options and show you how to implement each option in turn. The first option you'll be looking at is Lift's XML-based resource bundles.

12.2 Defining localized resources

Web applications present a rather different localization problem compared to desktop applications. The traditional Java view on localization is that you have a single per-locale bundle that contains all your application resources, and for desktop builds this often makes sense. But with web applications, this can become a little unwieldy, as it's not uncommon for web applications to have a moderately complex structure with lots of nested folders and pages and even page fragments. Suddenly, the Java approach of using a single properties bundle can become rather restrictive.

Although Lift can still support the traditional Java resource bundles (covered in section 12.2.2), Lift avoids their shortcomings and provides a richer method of localization through UTF-8 XML resource files.

12.2.1 Using XML resources

XML resource bundles can be located in a variety of places, but they primarily allow you to split the localized content up into logical sections: different bundles for each page or a more traditional global bundle for all pages. For example, assuming that the page URL is `/some/thing`, Lift will attempt to look for resources based upon `S.locale` with the following search path in your WAR file:

- 1 `webapp/some/_resources_thing`
- 2 `webapp/templates-hidden/some/_resources_thing`
- 3 `webapp/some/resources-hidden/_resources_thing`
- 4 `webapp/_resources`
- 5 `webapp/templates-hidden/_resources`
- 6 `webapp/resources-hidden/_resources`

As you can see, there are quite a number of places Lift will search for resources. The great thing here is that the first three locations provide you with page-specific resource bundles, and the last three provide global locations for common elements.

Now that you know where resources can be placed, consider the following listing, which shows an example of the Lift resource XML.

Listing 12.3 Example of Lift's resource XML

```
<?xml version="1.0"?>
<resources>
  <res name="greeting"
      lang="en"
      country="GB"
      default="true">Welcome!</res>
  <res name="greeting" lang="en" country="US">Howdy!</res>
  <res name="greeting" lang="fr">Bienvenue!</res>
  <res name="greeting" lang="de">Willkommen!</res>
  <res name="greeting" lang="it">Benvenuti!</res>
</resources>
```

Each `res` node (short for resource) has several possible attributes. The first is the `name` attribute, which defines the key for the resource you want to localize. The important difference here is that unlike many other localization systems, you can have the same key in the same resource set multiple times, provided their `country` and `lang` attributes define a different locale. In this example, you can see that there are three generic language resources for the `greeting` key: French, German, and Italian. These resources would be applied for any country that applies those languages. Conversely, there are two English implementations: British English and American English. Notice that the British English version also includes the `default` attribute, which essentially tells Lift that in the event of only having the language part of the locale, it should assume the proper British English.

Using this resource XML is a very Lift-specific way of doing localization, and although it's more flexible than traditional Java localization techniques, it's important to understand how you can use Java properties files if you so wish. The next section walks you through using Java properties files for localized resources.

12.2.2 Using Java properties resources

Because Lift runs on the JVM, it's straightforward to utilize the existing infrastructure for localizing your applications. Unlike the Lift XML resources, traditional properties resource bundles can only contain content for a single locale. The result of this is that you'll most likely end up with a directory in `src/main/resources/` that contains several files:

- `lift_de.properties`
- `lift_en_GB.properties`
- `lift_en_US.properties`
- `lift_it.properties`

The main drawback here is that all application resources for a given locale must exist in the same file, and that extended characters such as `œ` aren't properly encoded by default. This is due to Java properties bundles using ISO 8859-1 encoding and thus requiring conversion to escaped format, such as `\u0153`. Fortunately this process can be automated using tools like `native2ascii` but it's something else you need to think

about. On the plus side, if you're migrating from an existing Java application, your localized files will still operate exactly as they did.

NOTE Native2Ascii can be found at <http://download.oracle.com/javase/1.4.2/docs/tooldocs/windows/native2ascii.html>. There is also an SBT wrapper for native2ascii that autoconverts your raw text to escaped Unicode output. It can be found on GitHub: <https://github.com/timperrett/sbt-native2ascii-plugin>.

By default, the resource base name that Lift uses to locate your resource bundles is actually the word *lift*, meaning it will look for `lift_en.properties`, for example. You can override this and provide the following configuration in your `Boot` class to customize the resource base name:

```
LiftRules.resourceNames = "content" :: LiftRules.resourceNames
```

With this configuration, Lift will attempt to resolve resource bundles first with the base name *content* and then secondly with *lift*.

If neither Lift resource XML nor traditional Java properties localization methods are to your liking, you can always implement your own resource bundle factory to, for example, pull localized content from a database or any other kind of text store.

12.2.3 Using custom resource factories

Depending upon the type of application you're building, there may come a point when having all the resources externalized in static files isn't quite enough, and you'd like to plug in your own custom bundle factory. A bundle factory is a facility whereby you can provide your own class implementation that extends `java.util.ResourceBundle`, which essentially gives you the ability to pull the localized content from anywhere you can reach with that code; it's entirely up to you.

In order to implement a custom resource bundle factory, construct something similar to the following and yield your subtype of `ResourceBundle`:

```
LiftRules.resourceBundleFactories.append {
  case (key, locale) => // yield ResourceBundle
}
```

The partial function in this instance is passed a `Tuple2`, where the first element is the resource key and the second element is the specified locale instance. You can construct your own subtype of `ResourceBundle` in Scala and return it based upon this lookup. How you're actually storing the resource information will massively alter the lookup implementation, such as if you were storing the text in a NoSQL store versus using some kind of in-memory structure. The logic, however, would simply be to search the backing store (whatever that may be) looking for a key and locale that match the ones being passed to the partial function.

12.3 *Summary*

In this chapter, you've seen how to ready your application for the global market by leveraging Lift's sophisticated localization system to provide content to users in the manner to which they're accustomed. This included instructions on how to configure and optimize the locale calculator so your application can detect which locale it should apply for any given request. In addition, we also looked at how to bundle localized content and apply the content in both your template markup and application code.

The next chapter takes a look at a pair of subjects that are typically found in larger enterprise systems: concurrent programming models and integration with enterprise systems, making use of Lift's integration with JPA persistence. In addition, you'll be broadly introduced to the Akka distribution and concurrency toolkit to get a feel for how you could scale the backend of your Lift applications when building the next Twitter!

Lift IN ACTION

Timothy Perrett



Lift is a Scala-based web framework designed for extremely interactive and engaging web applications. It's highly scalable, production-ready, and will run in any servlet container. And Lift's convention-over-configuration approach lets you avoid needless work.

Lift in Action is a step-by-step exploration of the Lift framework. It moves through the subject quickly using carefully crafted, well-explained examples that make you comfortable from the start. You'll follow an entertaining Travel Auction application that covers the core concepts and shows up architectural and development strategies. Handy appendixes offer a Scala crash course and guidance for setting up a good coding environment.

What's Inside

- Complete coverage of the Lift framework
- Security, maintainability, and performance
- Integration and scaling
- Covers Lift 2.x

This book is written for developers who are new to both Scala and Lift and covers just enough Scala to get you started.

Timothy Perrett is a member of the Lift core team and a Scala developer specializing in integration and automation systems, for both manufacturing and marketing workflows.

For access to the book's forum and a free eBook for owners of this book, go to manning.com/LifinAction

“The best guide for building secure, scalable, and real-time web applications using Scala and Lift.”

—Guillaume Belrose
Quantel Ltd

“Rejoice, would-be Lift programmers, ... finally an approachable resource to turn to.”

—Ted Neward
Neward & Associates

“If you're building web apps with Lift, you need this book.”

—John C. Tyler, PROS Pricing

“Great reference book for Lift and Scala!”

—Tom Jensen
Chocolate Sprocket

ISBN 13: 978-1-93518-280-1
ISBN 10: 1-93518-280-3



9 781935 182801