Scala Cookbook: Bonus Chapters

Alvin Alexander

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Scala Cookbook: Bonus Chapters

by Alvin Alexander

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Courtney Nash
Production Editor: Rachel Steely
Copyeditor: Kim Cofer
Proofreader: Linley Dolby
Indexer: Ellen Troutman
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

August 2013: First Edition

Revision History for the First Edition:

2013-07-30: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449339616* for release details.

ISBN: 978-1-449-33961-6

[LSI]

Table of Contents

BONUS CHAPTER

XML and XPath

Introduction

Scala has built-in support for XML, so you can freely mix XML literals into your Scala source code. This is nice, because instead of using strings to represent XML, you create XML literals that the compiler can understand and verify. This approach lets you easily generate dynamic XML by interweaving Scala code and XML in the same expressions.

On the flip side, when it comes to searching XML for the data you need, you can use XPath methods like \ and \\ to find what you're looking for, and other methods like `child`, `label`, and `attributes` to iterate over elements and extract information.

As a word of caution, if you want to parse poorly formed XHTML, don't try to do this with the XML library. See my blog post for examples of how to use the Java HTMLCleaner library to solve this problem.

1. Creating XML Literals

Problem

You want to create XML variables, and embed XML into your Scala code.

Solution

You can assign XML expressions directly to variables, as shown in these examples:

```
val hello = <p>Hello, world</p>
val p = <person><name>Edward</name><age>42</age></person>
```

In the REPL, you can see that these variables are of type `scala.xml.Elem`:

```
scala> val hello = <p>Hello, world</p>
hello: scala.xml.Elem = <p>Hello, world</p>
```

```
scala> val p = <person><name>Edward</name><age>42</age></person>
p: scala.xml.Elem = <person><name>Edward</name><age>42</age></person>
```

As shown, you can embed XML directly into your Scala source code. There's no need to embed the XML in double quotes to create a `String`; just assign an XML block to a variable.

XML blocks can span multiple lines:

```
val foo = <p>Lorem ipsum
          dolor sit amet,
          consectetur adipisicing elit
          et cetera, et cetera</p>
```

A block can take as many lines as needed to solve the current problem. As shown in the following code, an XML literal can also be the result of a method or function:

```
def getXml =
  <pizza>
    <crust type="thin" size="14" />
    <topping>cheese</topping>
    <topping>sausage</topping>
  </pizza>
```

More often, you'll return dynamically generated XML from a method. I'll demonstrate that in Recipe 2.

Discussion

If you're given a block of XML as a `String`, you can convert it into an XML literal with the `loadString` method of the `scala.xml.XML` object:

```
scala> val dog = ↵
xml.XML.loadString("<dog><name>Rocky</name><age>12</age></dog>")
dog: scala.xml.Elem = <dog><name>Rocky</name><age>12</age></dog>
```

This results in an `Elem` object, as in the previous examples.

Note that poorly formed XML strings will lead to `SAXParseException`s:

```
scala> val x = scala.xml.XML.loadString("")
org.xml.sax.SAXParseException: Premature end of file.

scala> val x = scala.xml.XML.loadString("a")
org.xml.sax.SAXParseException: Content is not allowed in prolog.

scala> val x = scala.xml.XML.loadString("<a>")
org.xml.sax.SAXParseException: XML document structures must start
and end within the same entity.
```

The Scaladoc states that the `Elem` class “extends the `Node` class, providing an immutable data object representing an XML element.” An `Elem` has a label, attributes, and children, as you’ll see in many examples in this chapter.

When parsing XML, the classes you’ll run into most often are `Elem`, `Node`, and `NodeSeq`. The relationship between these classes (and a few others) is shown in Figure 1.

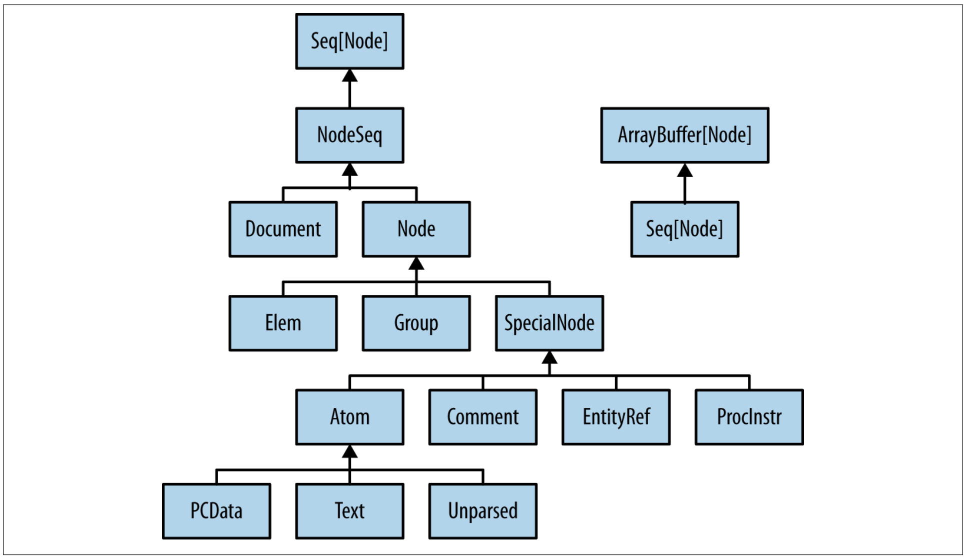


Figure 1. The main classes in the scala.xml class hierarchy

The `Elem`, `Node`, and `NodeSeq` classes are described in Table 1.

Table 1. Descriptions of the most commonly used scala.xml classes

Name	Description
`Elem`	An immutable object that represents an XML element.
`Node`	An abstract class representing nodes in an XML tree. It contains an implementation of XPath methods like \ and \\.
`NodeSeq`	A wrapper around `Seq[Node]`, with XPath and comprehension methods. Typically seen as the result of XPath searches.

Although the `Elem` class has more than 160 methods, don’t be intimidated; many of these come from the ability to treat an `Elem` as a sequential collection. There are a small handful of commonly used, XML-specific methods, and those are listed in Table 3 in Recipe 3.

The `NodeSeq` class is simpler. Like the `Elem` class, it implements the \ and \\ methods for XPath searching, and then is primarily composed of common collection methods.

If you get deeply involved in XML parsing and creation, you'll also run into some of the other Scala classes. The most common of those are listed in Table 2.

Table 2. Other scala.xml classes you may encounter

Name	Description
`NodeBuffer`	Extends `ArrayBuffer[Node]` and adds an &+ method that lets you build a sequence of XML nodes using a fluent style.
`PCData`	From the Scaladoc, "this class (which is not used by all XML parsers, but always used by the XHTML one) represents parseable character data, which appeared as CDATA sections in the input and is to be preserved as CDATA sections in the output." Example: `scala> val x = PCData("<p>hello</p>")` `x: scala.xml.PCData = <![CDATA[<p>hello</p>]]>`
`Text`	Implements an XML node for text (PCDATA). *Programming in Scala* refers to it as, "A node holding just text." Example: `scala> val x = Text("<p>Sundance</p>")` `x: scala.xml.Text = &lt;p&gt;Sundance&lt;/p&gt;`
`Unparsed`	An XML node for unparsed content. Per the Scaladoc, it will output verbatim, and "all bets are off regarding wellformedness etc." Example: `// intentional error` `scala> val x = Unparsed("</p>foo<p>")` `x: scala.xml.Unparsed = </p>foo<p>`

See Also

- The `Elem` class
- The `Node` class
- The `NodeSeq` class

2. Mixing Scala and XML

Problem

You want to dynamically generate XML from your Scala source code, such as creating output for a SOAP web service.

Solution

A great feature of Scala's XML support is that you can interweave XML and regular Scala source code together. This lets you dynamically generate XML from your Scala code.

To create XML with dynamic, embedded data, just put your Scala code in curly braces inside the XML tags, as shown in the following example:

```
scala> val name = "Bill"
name: String = Bill

scala> val age = 42
age: Int = 42

scala> val p = <person><name>{name}</name><age>{age}</age></person>
p: scala.xml.Elem = <person><name>Bill</name><age>42</age></person>
```

In this example, the Scala variables `name` and `age` are put inside curly braces, inside the XML literals. The variable `p` in the REPL results shows that they're translated to their values (`Bill` and `42`, respectively). Methods and other expressions can be used in the same way.

This ability to weave Scala code and XML together is similar to using a templating system, and is a great way to generate dynamic XML output, including output like an RSS news feed, all forms of business data, or a simple XHTML `UL`/`LI` tag combination:

```
scala> val fruits = List("apple", "banana", "orange")
fruits: List[java.lang.String] = List(apple, banana, orange)

scala> val ul = <ul>{fruits.map(i => <li>{i}</li>)}</ul>
ul: scala.xml.Elem = <ul><li>apple</li><li>banana</li><li>orange</li></ul>

scala> println(ul)
<ul><li>apple</li><li>banana</li><li>orange</li></ul>
```

You can use the same technique to generate XHTML `<select>` and `<option>` tags, such as for a list of states or credit card options, and any other form of XML data.

Discussion

As shown in the `fruits` example, XML literals can contains Scala code in curly braces, and that code can include additional XML literal values, which can contain Scala code. This nesting of Scala and XML can continue to go deeper, as needed.

To explain how this works, let's look at the code again:

```
val ul = <ul>{fruits.map(i => <li>{i}</li>)}</ul>
```

Here's how this code works:

- The XML expression is enclosed in the matching `<ul>` tags.
- The curly braces begin the `fruits.map` Scala expression.
- `<li>` tags are embedded in the code block passed to the `map` method.
- The variable `i` is enclosed in curly braces inside the `<li>` tags, where it will be replaced by its values as the `map` method executes.

If you've used XML/HTML templating tools previously, you can appreciate the power of this approach.

NodeBuffer

The `NodeBuffer` class provides another nice way to dynamically build XML. The following example shows how to build a set of `<li>` tags as a `NodeBuffer`, and then insert those elements into a final `<ul>` tag:

```
scala> val x = new xml.NodeBuffer
x: scala.xml.NodeBuffer = ArrayBuffer()

scala> x += <li>apple</li>
res0: x.type = ArrayBuffer(<li>apple</li>)

scala> x += <li>banana</li>
res1: x.type = ArrayBuffer(<li>apple</li>, <li>banana</li>)

scala> val ul = <ul>{x}</ul>
ul: scala.xml.Elem = <ul><li>apple</li><li>banana</li></ul>
```

`NodeBuffer` is a simple convenience class that extends `ArrayBuffer[Node]`. It adds one method named `&+` that appends the given object to the buffer and returns a `this` reference for convenience. This lets you write a "fluent" style of code like this, if you prefer:

```
val nb = new xml.NodeBuffer
val nb2 = nb &+ <li>apple</li> &+ <li>banana</li> &+ <li>cherry</li>
```

3. Extracting Data from XML Nodes

Problem

You want to extract information from XML you receive, so you can use the data in your application.

Solution

Use the methods of Scala `Elem` and `NodeSeq` classes to extract the data. The most commonly used methods of the `Elem` class are shown in Table 3.

Table 3. Commonly used methods of the Elem class

Method	Description
`x \ "div"`	Searches the XML literal x for elements of type `<div>`. Only searches immediate child nodes (no grandchild or "descendant" nodes).
`x \\ "div"`	Searches the XML literal x for elements of type `<div>`. Returns matching elements from child nodes at *any* depth of the XML tree.
`x.attribute("class")`	Returns the value of the given attribute in the current node. `<a x="10" y="20">foo</a>.attribute("x")` returns `Some(10)`.

Method	Description

`x.attributes`	Returns all attributes of the current node, prefixed and unprefixed, in no particular order.
	`scala> <a x="10" y="20">foo</a>.attributes`
	`res0: scala.xml.MetaData = x="10" y="20"`

`x.child`	Returns the children of the current node. `<a><b>foo</b></a>.child` returns `<b>foo</b>`.
`x.copy(...)`	Returns a copy of the element, letting you replace data during the copy process.
`x.label`	The name of the current element. `<a><b>foo</b></a>.label` returns `a`.
`x.text`	Returns a concatenation of `text(n)` for each child `n`.
`x.toString`	Emits the XML literal as a `String`. Use `scala.xml.PrettyPrinter` to format the output, if desired.

The following examples demonstrate most of the methods in Table 3. Given this XML literal:

```
scala> val x = <div class="content"><p>Hello</p><p>world</p></div>
x: scala.xml.Elem = <div class="content"><p>Hello</p><p>world</p></div>
```

you can search for and extract subelements with the \ and \\ XPath methods:

```
scala> x \ "p"
res0: scala.xml.NodeSeq = NodeSeq(<p>Hello</p>, <p>world</p>)

scala> x \\ "p"
res1: scala.xml.NodeSeq = NodeSeq(<p>Hello</p>, <p>world</p>)
```

These methods will be demonstrated more in subsequent recipes.

The `label` method returns the name of the current element. A `<p>` tag returns `p`, a `<div>` tag returns `div`, etc.:

```
scala> x.label
res2: String = div

scala> <name>Joe</name>.label
res3: String = name
```

The `text` method returns the text from all subelements, which the Scaladoc describes as “a concatenation of all `text(n)` for each child `n`”:

```
scala> x.text
res4: String = Helloworld
```

Later examples will demonstrate how to improve on this result.

Element attributes are extracted with the `attribute` or `attributes` methods. The following examples demonstrate how to call these methods, and the values they return:

```
scala> x.attribute("class")
res5: Option[Seq[scala.xml.Node]] = Some(content)
```

```
scala> x.attributes("class")
res6: Seq[scala.xml.Node] = content

scala> x.attributes.get("class")
res7: Option[Seq[scala.xml.Node]] = Some(content)
```

The following examples demonstrate how those same method calls behave when you search for an attribute that doesn't exist:

```
scala> x.attribute("foo")
res8: Option[Seq[scala.xml.Node]] = None

scala> x.attributes("foo")
res9: Seq[scala.xml.Node] = null

scala> x.attributes.get("foo")
res10: Option[Seq[scala.xml.Node]] = None

scala> x.attributes.get("foo").getOrElse("N/A")
res11: Object = N/A
```

To demonstrate more ways to work with element attributes, let's create a new element:

```
scala> val w = <forecast day="Thu" date="10 Nov 2011" low="37" high="58" />
w: scala.xml.Elem = <forecast day="Thu" date="10 Nov 2011" low="37" high="58" />
```

These examples show how `attribute` and `attributes` work with multiple attributes:

```
scala> w.attribute("day")
res0: Option[Seq[scala.xml.Node]] = Some(Thu)

scala> w.attributes("day")
res1: Seq[scala.xml.Node] = Thu

scala> w.attributes
res2: scala.xml.MetaData =  day="Thu" date="10 Nov 2011" low="37" high="58"
```

These examples show how to iterate over a set of attributes:

```
scala> for (a <- w.attributes) println(s"key: ${a.key}, value: ${a.value}")
key: day, value: Thu
key: date, value: 10 Nov 2011
key: low, value: 37
key: high, value: 58

scala> w.attributes.asAttrMap
res3: Map[String,String] = Map(low -> 37, date -> 10 Nov 2011,
      day -> Thu, high -> 58)
```

Child elements

The `child` method returns all child nodes of the current element. To demonstrate this, let's create a new XML variable:

```
scala> val p = <person><name>Ken</name><age>23</age></person>
p: scala.xml.Elem = <person><name>Ken</name><age>23</age></person>
```

The `child` method returns immediate child nodes:

```
scala> p.child
res0: Seq[scala.xml.Node] = ArrayBuffer(<name>Ken</name>, <age>23</age>)
```

You can use `child` to iterate over all the children:

```
scala> for (n <- p.child) println(n)
<name>Ken</name>
<age>23</age>
```

Because `child` returns a sequence, you can also access the child elements like this:

```
scala> p.child(0)
res1: scala.xml.Node = <name>Ken</name>

scala> p.child(0).label
res2: String = name

scala> p.child(0).text
res3: String = Ken

scala> p.child(1)
res4: scala.xml.Node = <age>23</age>

scala> p.child(1).text.toInt
res5: Int = 23
```

Text and strings

The `toString` method returns the XML structure as a `String`:

```
scala> p.toString
res6: String = <person><name>Ken</name><age>23</age></person>
```

You can improve this result with the `PrettyPrinter`. See Recipe 11.

This approach shows another way to extract the text from the elements:

```
scala> for (n <- p.child) yield n.text
res7: Seq[String] = ArrayBuffer(Ken, 23)
```

There are more ways to tackle these problems using XPath methods, which will be shown in subsequent chapters.

As a word of caution, be careful with the `text` method. It returns different results depending on how the XML is formatted, which can be a particular problem when extracting XHTML data. To demonstrate this, the following examples show the output when there is a space before the `<br/>` tag, and when there is no space:

```
scala> <div><p>Hello, world, <br/>it's me.</p></div>.text
res0: String = Hello, world, it's me.
```

```
scala> <div><p>Hello, world,<br/>it's me.</p></div>.text
res1: String = Hello, world,it's me.
```

In the next examples, the same XML, formatted in different ways, yields different results:

```
scala> <div><p>Is 2 &gt; 1?</p><p>Why do you ask?</p></div>.text
res2: String = Is 2 > 1?Why do you ask?

scala> <div>
     | <p>Is 2 &gt; 1?</p>
     | <p>Why do you ask?</p>
     | </div>.text
res3: String =
"
Is 2 > 1?
Why do you ask?
"
```

If you need to extract text in this manner, a workaround is to extract the text components individually into a sequence, and then recombine the sequence as desired. The following example demonstrates how to accomplish this with the `child`, `label`, and `text` methods. Given this XML literal:

```
val xml = <div><p>Is 2 &gt; 1?</p><p>Why do you ask?</p></div>
```

the `child` method returns the elements as a sequence:

```
scala> xml.child
res0: Seq[scala.xml.Node] =
  ArrayBuffer(<p>Is 2 &gt; 1?</p>, <p>Why do you ask?</p>)
```

This lets you write the following code, which creates a sequence of strings from the `<p>` tags:

```
val strings = for {
  e <- xml.child
  if e.label == "p"
} yield e.text
```

The REPL shows that the resulting variable `strings` has the following type and data:

```
strings: Seq[String] = ArrayBuffer(Is 2 > 1?, Why do you ask?)
```

In the XPath recipes in this chapter, you'll see how to accomplish some of the same tasks using the `\` and `\\` methods.

Example data sets and REPL memory errors

If you want to test these commands against large data sets, the University of Washington's Computer Science department maintains a nice collection of sample XML data.

The NASA data set is 23 MB, and causes the REPL to crash with a Java heap space error:

```
scala> val xml = scala.xml.XML.loadFile("nasa.xml")
java.lang.OutOfMemoryError: Java heap space ...
```

To get around this problem, you can allocate more heap space when starting the REPL with this command:

```
$ scala -J-Xms256m -J-Xmx512m
```

or this command:

```
$ env JAVA_OPTS="-Xms256m -Xmx512m" scala
```

4. Basic XPath Searching with \ and \\

Problem

You want to search an XML tree for the data you need using XPath expressions.

Solution

Use the `\` and `\\` methods, which are analogous to the XPath `/` and `//` expressions. The `\` method returns all matching elements *directly* under the current node, and `\\` returns all matching elements from *all* nodes under the current node (all descendant nodes).

To demonstrate this difference, create this XML literal:

```
scala> val a = <div><p>Hello,<br/>world</p></div>
a: scala.xml.Elem = <div><p>Hello,<br/>world</p></div>
```

The `\` method finds the `<p>` tag because it's directly under the `<div>` tag:

```
scala> a \ "p"
res0: scala.xml.NodeSeq = NodeSeq(<p>Hello,<br/>world</p>)
```

But it won't find the `<br/>` tag in the XML literal, because it's not directly under the `<div>` tag:

```
scala> a \ "br"
res1: scala.xml.NodeSeq = NodeSeq()
```

However, the `\\` method can find it, because it searches through all descendant nodes (children, grandchildren, etc.) under the `<div>` tag:

```
scala> a \\ "br"
res2: scala.xml.NodeSeq = NodeSeq(<br/>)
```

Using the \ method

As a deeper demonstration of the `\` method, first create this XML literal:

```
val x = <stocks>
  <stock>AAPL</stock>
  <stock>AMZN</stock>
  <stock>GOOG</stock>
</stocks>
```

Given this XML, the following expression returns all `<stock>` elements:

```
scala> x \ "stock"
res0: scala.xml.NodeSeq =
  NodeSeq(<stock>AAPL</stock>, <stock>AMZN</stock>, <stock>GOOG</stock>)
```

As shown in the REPL output, this returns an instance of a `NodeSeq`, which is a simple wrapper around `Seq[Node]` (a sequence of nodes). Like the `Elem` class, `NodeSeq` supports the \ and \\ search methods, as well as the usual large variety of collection methods.

Once you have a `NodeSeq`, you can work with the data it contains. For instance, you can create a list of stock symbols like this:

```
scala> (x \ "stock").map(_.text)
res1: scala.collection.immutable.Seq[String] = List(AAPL, AMZN, GOOG)
```

If this is confusing, it can help to see it broken down into smaller steps. First, get a sequence of elements with the \ method, and assign the result to a variable:

```
scala> val nodes = x \ "stock"
nodes: scala.xml.NodeSeq =
  NodeSeq(<stock>AAPL</stock>, <stock>AMZN</stock>, <stock>GOOG</stock>)
```

You can see that `nodes` is a variable of type `NodeSeq`. Each individual node is of type `Elem`:

```
scala> for (n <- nodes) println(n.getClass)
class scala.xml.Elem
class scala.xml.Elem
class scala.xml.Elem
```

Each element contains its XML tag as well as its data:

```
scala> for (n <- nodes) println(n)
<stock>AAPL</stock>
<stock>AMZN</stock>
<stock>GOOG</stock>
```

So, to extract only the data from each node, call the `text` method:

```
scala> for (n <- nodes) println(n.text)
AAPL
AMZN
GOOG
```

Putting this together, you can create a list of stock names using a `for/yield` loop:

```
scala> val stockNames = for (n <- nodes) yield n.text
stockNames: scala.collection.immutable.Seq[String] = List(AAPL, AMZN, GOOG)
```

That loop is equivalent to this `map` method call:

```
scala> val stockNames = nodes.map(_.text)
stockNames: scala.collection.immutable.Seq[String] = List(AAPL, AMZN, GOOG)
```

Because `NodeSeq` has all the usual sequence methods, it's easy to get the information you want from the XML. For instance, you can find the number of nodes, or filter the results to get only the stocks you want:

```
// same as 'nodes.length'
scala> (x \ "stock").length
res1: Int = 3

scala> nodes.map(_.text).filter(_.startsWith("A"))
res2: scala.collection.immutable.Seq[String] = List(AAPL, AMZN)
```

Using the \\ method

The \ method only returns matches on immediate subelements. To search deeper—the entire XML tree—use the \\ method.

Given this XML:

```
val x =
<portfolio>
  <stocks>
    <stock>AAPL</stock>
    <stock>AMZN</stock>
    <stock>GOOG</stock>
  </stocks>
  <reits>
    <reit>Super REIT 1</reit>
  </reits>
</portfolio>
```

the \ method returns an empty `NodeSeq` when searching for `<stock>` elements:

```
scala> x \ "stock"
res0: scala.xml.NodeSeq = NodeSeq()
```

You can solve this problem by exactly specifying the path to the `<stock>` elements with multiple \ method calls:

```
scala> x \ "stocks" \ "stock"
res1: scala.xml.NodeSeq =
  NodeSeq(<stock>AAPL</stock>, <stock>AMZN</stock>, <stock>GOOG</stock>)
```

But the \\ method can be a simpler approach to finding the desired elements. It searches the entire XML tree to find all elements that match your search query:

```
scala> (x \\ "stock").foreach(println)
<stock>AAPL</stock>
<stock>AMZN</stock>
<stock>GOOG</stock>
```

As shown before, you can convert the XML data to a list of strings, if desired:

```
scala> (x \\ "stock").map(_.text)
res2: scala.collection.immutable.Seq[String] = List(AAPL, AMZN, GOOG)
```

Discussion

In addition to the approaches shown, you can use the _ wildcard with the \ and \\ methods. For instance, given this XML that represents a group of people you know:

```
val people =
<people>
  <family>
    <person>Mom</person>
  </family>
  <friends>
    <person>Bill</person>
    <person>Candy</person>
  </friends>
</people>
```

You can list family members like this:

```
scala> val family = people \ "family" \ "person"
family: scala.xml.NodeSeq = NodeSeq(<person>Mom</person>)
```

You can list friends like this:

```
scala> val friends = people \ "friends" \ "person"
friends: scala.xml.NodeSeq =
NodeSeq(<person>Bill</person>, <person>Candy</person>)
```

You can list everyone you know by using the _ wildcard in place of specifying `family` or `friends`:

```
scala> val allPeople = people \ "_" \ "person"
allPeople: scala.xml.NodeSeq =
NodeSeq(<person>Mom</person>, <person>Bill</person>, <person>Candy</person>)
```

Without the wildcard character, you'd have to create the lists of family and friends and then merge them together manually.

Once you have the list of people, you can access the elements one at a time:

```
scala> allPeople(0)
res0: scala.xml.Node = <person>Mom</person>

scala> allPeople(1)
res1: scala.xml.Node = <person>Bill</person>
```

You can also iterate over all of the elements as usual:

```
scala> allPeople.foreach(println)
<person>Mom</person>
<person>Bill</person>
<person>Candy</person>

scala> for (person <- allPeople) println(person.text)
Mom
Bill
```

```
Candy

scala> allPeople.map(_.text)
res2: scala.collection.immutable.Seq[String] = List(Mom, Bill, Candy)
```

For more examples of how to use the \ and \\ methods, see Recipe 5, "Deeper XML Parsing and Extracting Tag Attributes".

See Also

- The `Elem` class
- The `NodeSeq` class

5. Deeper XML Parsing and Extracting Tag Attributes

Problem

You need to perform deep XML searches, combining the \ and \\ methods, and possibly searching directly for tag attributes.

Solution

Combine the \\ and \ methods as needed to search the XML. When you need to extract tag attributes, place an `@` character before the attribute name.

Given this simplified version of the Yahoo Weather RSS Feed:

```
val weather =
<rss>
  <channel>
    <title>Yahoo! Weather - Boulder, CO</title>
    <item>
     <title>Conditions for Boulder, CO at 2:54 pm MST</title>
     <forecast day="Thu" date="10 Nov 2011" low="37" high="58" text="Partly Cloudy"
               code="29" />
    </item>
  </channel>
</rss>
```

you can access the `<forecast>` tag like this:

```
scala> val forecast = weather \ "channel" \ "item" \ "forecast"
forecast: scala.xml.NodeSeq = NodeSeq(<forecast day="Thu"
  date="10 Nov 2011" low="37" high="58" text="Partly Cloudy" code="29"/>)
```

You can also directly access the attributes of the `<forecast>` element with these expressions:

```
val day = weather \ "channel" \ "item" \ "forecast" \ "@day"
val date = weather \ "channel" \ "item" \ "forecast" \ "@date"
```

However, once you've created the `forecast` variable, it's easier to access the attributes like this:

```
val day = forecast \ "@day"     // Thu
val date = forecast \ "@date"   // 10 Nov 2011
val low = forecast \ "@low"     // 37
val high = forecast \ "@high"   // 58
val text = forecast \ "@text"   // Partly Cloudy
```

Each of these attributes is returned as a `NodeSeq`:

```
scala> val day = forecast \ "@day"
day: scala.xml.NodeSeq = Thu
```

You can convert that to a `String` with the `text` method:

```
scala> val day = (forecast \ "@day").text
day: String = Thu
```

A nice feature of this approach is that if an attribute is missing, it kindly returns an empty `NodeSeq`:

```
scala> val foo = forecast \ "@foo"
foo: scala.xml.NodeSeq = NodeSeq()
```

This makes it easy to iterate over the results when elements are found, and when they're not found.

I created the `forecast` variable by specifying the full path to the `<forecast>` tag attributes, but you can simplify the expression by using `\\` instead of `\`:

```
scala> val day = weather \\ "forecast" \ "@day"
day: scala.xml.NodeSeq = Thu
```

If you're comfortable with your data—for instance, if you know there is only one `day` attribute that can be found—you can shorten that expression to only this:

```
scala> val day = weather \\ "@day"
day: scala.xml.NodeSeq = NodeSeq(Thu)
```

Discussion

To demonstrate more XPath search expressions, create this XML literal:

```
val xml =
  <order>
    <item name="Pizza" price="12.00">
      <pizza>
        <crust type="thin" size="14" />
        <topping>cheese</topping>
        <topping>sausage</topping>
      </pizza>
    </item>
    <item name="Breadsticks" price="4.00">
      <breadsticks />
    </item>
    <tax type="federal">0.80</tax>
    <tax type="state">0.80</tax>
    <tax type="local">0.40</tax>
  </order>
```

The following examples, which combine XPath and Scala expressions, show how to extract different pieces of information from that XML, including the elements and attributes. The comments before each expression state what I'm looking for:

```
// get the <item> elements from the order
scala> val items = xml \ "item"
items: scala.xml.NodeSeq =
NodeSeq(<item name="Pizza" price="12.00">
      <pizza>
        <crust type="thin" size="14"/>
        <topping>cheese</topping>
        <topping>sausage</topping>
      </pizza>
    </item>, <item name="Breadsticks" price="4.00">
      <breadsticks/>
    </item>)

// number of items in the order
scala> val numItems = items.length
numItems: Int = 2

// list of item prices
scala> val prices = items.map(i => i \ "@price")
prices: scala.collection.immutable.Seq[scala.xml.NodeSeq] = List(12.00, 4.00)

// the subtotal price
scala> val subtotal = items.map(i => (i \ "@price").text.toDouble).sum
subtotal: Double = 16.0

// list of taxes
scala> val taxItems = xml \ "tax"
taxItems: scala.xml.NodeSeq = NodeSeq(<tax type="federal">0.80</tax>,
<tax type="state">0.80</tax>, <tax type="local">0.40</tax>)

// the total tax
scala> val totalTax = taxItems.map(i => i.text.toDouble).sum
totalTax: Double = 2.0

// list of toppings on the pizza
scala> val toppings = (item \ "pizza" \ "topping").map(_.text)
toppings: scala.collection.immutable.Seq[String] = List(cheese, sausage)
```

You can access individual tax items like this:

```
// get the federal tax
val federalTax = for {
  item <- taxItems
  if (item \ "@type").text == "federal"
} yield item.text
```

That code returns a `List(0.80)`, a `List[String]`, which you can convert to a numeric value as shown in the examples.

6. Extracting Data from an Array of XML Elements

Problem

Your XML data has an array of elements, and you need to extract the first element, second element, or more generally, the Nth element.

Solution

The following simplified version of the XML from the Yahoo Weather API has three `<forecast>` elements:

```
val weather = <rss>
<channel>
<title>Yahoo! Weather - Boulder, CO</title>
<item>
<!-- multiple yweather:forecast elements -->
<forecast day="Thu" date="10 Nov 2011" low="37" high="58"
          text="Partly Cloudy" code="29" />
<forecast day="Fri" date="11 Nov 2011" low="39" high="58"
          text="Mostly Cloudy" code="28" />
<forecast day="Sat" date="12 Nov 2011" low="32" high="49" text="Cloudy"
          code="27" />
</item>
</channel>
</rss>
```

To access the data in the first `<forecast>` element, wrap the XPath expression in parentheses and append `(0)` to it. You can access the first element using a series of `\` method calls:

```
val day = (weather \ "channel" \ "item" \ "forecast")(0) \ "@day"
val date = (weather \ "channel" \ "item" \ "forecast")(0) \ "@date"
```

Or you can access it with a single `\\` method call, if you prefer:

```
val low = (weather \\ "forecast")(0) \ "@low"
val high = (weather \\ "forecast")(0) \ "@high"
```

Either approach yields this result:

```
scala> val date = (weather \\ "forecast")(0) \ "@date"
date: scala.xml.NodeSeq = 10 Nov 2011
```

Better yet, create a `forecasts` object first, and then extract the attributes from it:

```
// 1) creates a NodeSeq with the three <forecast> elements
val forecasts = weather \ "channel" \ "item" \ "forecast"

// 2) extract the attributes
val day  = forecasts(0) \ "@day"    // Thu (as a NodeSeq)
val date = forecasts(0) \ "@date"   // 10 Nov 2011
val low  = forecasts(0) \ "@low"    // 37
val high = forecasts(0) \ "@high"   // 58
val text = forecasts(0) \ "@text"   // Partly Cloudy
```

This approach returns the elements as a `NodeSeq`:

```
scala> val day = forecasts(0) \ "@day"
day: scala.xml.NodeSeq = Thu
```

To extract the attributes as a `String` instead, add the `text` method to the end of the expression:

```
scala> val day = (forecasts(0) \ "@day").text
day: String = Thu
```

If the attribute doesn't exist, this returns an empty string:

```
scala> val foo = ((weather \\ "forecast")(0) \ "@FOO").text
foo: String = ""
```

You can access data from other `<forecast>` elements in the same way. Here's the date from the second element in the array:

```
scala> val date = ((weather \\ "forecast")(1) \ "@date").text
date: String = 11 Nov 2011
```

As with any array you need to be careful, because if you try to access an array element that doesn't exist, you'll get an `IndexOutOfBoundsException`:

```
scala> val date = ((weather \\ "forecast")(49) \ "@date").text
java.lang.IndexOutOfBoundsException: 49
```

Iterating over the elements

If instead of accessing the `<forecast>` nodes as individual array elements, you want to handle the same data in a loop, first grab all of the `<forecast>` nodes using an XPath expression, and then iterate over them, as desired:

```
val forecastNodes = (weather \\ "forecast")

forecastNodes.foreach{ n =>
  val day  = (n \ "@day").text
  val date = (n \ "@date").text
  val low  = (n \ "@low").text
```

```
    println(s"$day, $date, Low: $low")
  }
```

This results in the following output:

```
Thu, 10 Nov 2011, Low: 37
Fri, 11 Nov 2011, Low: 39
Sat, 12 Nov 2011, Low: 32
```

Discussion

To explain this approach, it helps to see that when accessing array elements by their index value, the first portion of the search finds the `<forecast>` elements and returns them as a `NodeSeq`:

```
scala> weather \\ "forecast"
res0: scala.xml.NodeSeq = NodeSeq(
<forecast high="58" low="37" day="Thu" code="29" date="10 Nov 2011"
          text="Partly Cloudy"></forecast>,
<forecast high="58" low="39" day="Fri" code="28" date="11 Nov 2011"
          text="Mostly Cloudy"></forecast>,
<forecast high="49" low="32" day="Sat" code="27" date="12 Nov 2011"
          text="Cloudy"></forecast>)
```

Enclosing the expression in parentheses and adding `(0)` after it returns the zeroth element of the array:

```
scala> (weather \\ "forecast")(0)
res1: scala.xml.Node = <forecast high="58" low="37" day="Thu" code="29"
date="10 Nov 2011" text="Partly Cloudy"></forecast>
```

Each element in the `NodeSeq` is an `Elem` instance:

```
scala> (weather \\ "forecast")(0).getClass
res0: Class[_ <: scala.xml.Node] = class scala.xml.Elem
```

Therefore, once you're working with one `<forecast>` element, you can access its tag attributes, such as the `day` attribute:

```
scala> (weather \\ "forecast")(0) \ "@day"
res1: scala.xml.NodeSeq = Thu
```

As with any array, add `(1)`, `(2)`, etc., to access the other `<forecast>` elements.

See Also

The Yahoo Weather API

7. Using Match Expressions with XML

Problem

You want to use match expressions as another way to access the information contained in XML data.

Solution

Given this XML literal:

```
val pizzaNode =
  <pizza>
    <crust type="thin" size="14" />
    <topping>cheese</topping>
    <topping>sausage</topping>
  </pizza>
```

you can access the `<topping>` and `<crust>` elements with a match expression, like this:

```
pizzaNode match {
  case <topping>{value}</topping> => println(s"Got a topping: $value")
  case <crust /> => println("Got a <crust/> tag")
  case _ => println("D'oh!")
}
```

You'll usually put a match expression like this in a method, so let's do that here:

```
/**
* Version 1
* A pizza node can have <topping> and <crust> tags.
*/
def handlePizzaNode(pizzaNode: Node) {
  pizzaNode match {
    case <topping>{value}</topping> => println(s"Got a topping: $value")
    case <crust /> => println("Got a <crust/> tag")
    case _ => println("D'oh!")
  }
}
```

A few examples in the REPL demonstrate how this works. First, a `<topping>` element:

```
scala> val node = <topping>cheese</topping>
node: scala.xml.Elem = <topping>cheese</topping>

scala> handlePizzaNode(node)
Got a topping: cheese
```

Next, a `<crust>` element:

```
scala> val node = <crust type="thin" size="14" />
node: scala.xml.Elem = <crust type="thin" size="14"/>
```

```
scala> handlePizzaNode(node)
Got a <crust/> tag
```

The following example demonstrates how to iterate over all the `<topping>` nodes in the `<pizza>`:

```
scala> for (topping <- pizzaNode \ "topping") handlePizzaNode(topping)
Got a topping: cheese
Got a topping: sausage
```

Use @ to access node attributes

Accessing the attributes of the `<crust>` tag and using them on the right side of its `case` statement takes a bit more work.

As a solution to this problem, add a variable to a pattern match:

```
variableName @ pattern
```

This creates a variable-binding pattern. If the pattern succeeds, it sets the variable to the object it matches.

Using this approach to solve the current problem, rewrite the `<crust>` tag `case` statement like this:

```
case crust @ <crust /> => println(s"Got a <crust/> tag: $crust")
```

The modified version of the method now looks like this:

```
/**
* Version 2: Get access to the <crust> tag attributes.
*/
def handlePizzaNode(pizzaNode: Node) {
  pizzaNode match {
    case <topping>{value}</topping> =>
           println(s"Got a topping: $value")
    case crust @ <crust /> =>
           val crustSize = crust \ "@size"
           val crustType = crust \ "@type"
           println(s"crustSize: $crustSize, crustType: $crustType")
    case _ =>
           println("D'oh!")
  }
}
```

Running the code in the REPL again, you can now access the `<crust>` data on the right side of the `case` statement:

```
scala> val node = <crust type="thin" size="14" />
node: scala.xml.Elem = <crust type="thin" size="14"/>

scala> handlePizzaNode(node)
crustSize: 14, crustType: thin
```

A real-world method should either return the nodes that were found by each case match, or call a method to act on each node. The following code demonstrates the second approach, how you might act on each node by calling addTopping, setCrustSize, and setCrustType methods (which are left as an exercise for the reader to implement):

```
/**
* Version 3: Call useful methods
*/
def handlePizzaNode(pizzaNode: Node) {
  pizzaNode match {
    case <topping>{value}</topping> =>
         addTopping(value)
    case crust @ <crust /> =>
         setCrustSize((node \ "@size").text)
         setCrustType((node \ "@type").text)
    case _ => println("D'oh!")
  }
}
```

Handling an array of elements

Now that you've seen the use of the @ symbol in case statements, you can extend the approach to handle multiple elements of the same type, such as multiple <stock> elements contained in this XML literal:

```
val stocks =
  <stocks>
    <stock>AAPL</stock>
    <stock>AMZN</stock>
    <stock>GOOG</stock>
  </stocks>
```

The following match expression shows the formula to access and print the value of each <stock> element:

```
stocks match {
  case <stocks>{stocks @ _*}</stocks> =>
    for (stock @ <stock>{_*}</stock> <- stocks)
      println(s"stock: ${stock.text}")
}
```

When that code is run, it yields the following output:

```
stock: AAPL
stock: AMZN
stock: GOOG
```

Discussion

There are a few pitfalls to be aware of when using match expressions with XML. For example, I showed an earlier example like this that works:

```
// this works
<p>Hello, world</p> match {
  case <p>{text}</p> => println(text)
}
```

But it's important to know that in a match expression, the tags have to be an *exact* match. The following match expression blows up with a `MatchError`, because I'm looking for text between matching `<topping>` tags, but the text is given in an attribute of the `<topping>` tag:

```
// throws a MatchError
<topping value="Hello, world" /> match {
  case <topping>{text}</topping> => println(text)
}
```

Although that demonstrates a communication error that's likely to throw off most algorithms, you'll also get a `MatchError` if the `<topping>` tags are formed correctly, but are empty:

```
// throws a MatchError
<topping></topping> match {
  case <topping>{text}</topping> => println(text)
}
```

With that `case` statement, you'll also get a `MatchError` if the `<topping>` tags contain another XML tag:

```
// throws a MatchError
<topping>Green<br/>olives</topping> match {
  case <topping>{text}</topping> => println(text)
}
```

To solve these problems, and other similar problems, the correct approach is to write one or more `case` statements to handle the XML you want to allow, and then add a default `case` statement to prevent a `MatchError` from being thrown by your match expression, as shown here:

```
<topping></topping> match {
  case <topping>{text}</topping> => println(text)
  case _ => println("got something else")
}
```

Handling unexpected tags

XML tags embedded in other XML tags can also cause problems in match expressions. I showed in other recipes that you can extract the text from the following XML literal, even though it contains a `<br/>` tag:

```
scala> <p>mystery<br/></p>.text
res0: String = mystery
```

But attempting to match that XML literal in the following match expression throws a `MatchError`:

```
// this throws a MatchError
<p>Hello, <br/>world</p> match {
  case <p>{content}</p> => println(content)
}
```

A quick trip into the REPL helps us understand the problem:

```
scala> val hello = <p>Hello, <br/>world</p>
hello: scala.xml.Elem = <p>Hello, <br/>world</p>

scala> hello.child.foreach(e => println(e.getClass))
class scala.xml.Text
class scala.xml.Elem
class scala.xml.Text
```

`hello` was expected to contain plain text, but it contains other XML entities. Contrast that output with the output from this example, where the `<br/>` tag is removed:

```
scala> val hello = <p>Hello, world</p>
hello: scala.xml.Elem = <p>Hello, world</p>

scala> hello.child.foreach(e => println(e.getClass))
class scala.xml.Text
```

When only `<p>` tags are used, `hello` contains only a `Text` element, and you can successfully get a match in your match expression; but when one or more `<br/>` tags are included inside the `<p>` tags, the match expression fails.

To solve this problem, you need to change the left side of the `case` statement. Using the `@` approach shown in the Solution helps you get where you need to be:

```
// this works
<p>Hello, <br/>world</p> match {
  case <p>{ n @ _* }</p> => n.foreach(println)
}
```

This prints the following output in the REPL:

```
Hello,
<br/>
world
```

In summary, use this syntax to handle the left side of the `case` statement, and manipulate the resulting elements as needed on the right side of the statement.

Can't match attributes

As shown in the Solution, you can't easily match node attributes in a match expression. To get around this problem, use one of the following approaches with your `case` statements.

If your XML node only has one attribute, like this:

```
val node = <crust type="thin" />
```

you can use this approach:

```
node match {
  case c @ <crust/> if (c \ "@type").text == "thin" => println(s"type is thin")
  case c @ <crust/> if (c \ "@type").text == "thick" => println(s"type is thin")
}
```

With multiple attributes, you'll probably want to use the approach shown in the Solution, handling the attributes on the right side of the `case` statement:

```
val node = <crust type="thin" size="14" />

node match {
  case crust @ <crust /> =>
       val crustSize = crust \ "@size"
       val crustType = crust \ "@type"
       println(s"crustSize: $crustSize, crustType: $crustType")
}
```

See Also

> The pattern matching approach shown in this recipe is similar to other recipes in this chapter, and can also be combined with them, specifically Recipes 3 through 6.

8. Serializing and Deserializing XML

Problem

You want to convert the data in a Scala class to its equivalent XML, or convert an XML representation of a class to an instance of the class.

Solution

There are two primary approaches to solving this problem:

- Convert the fields in your Scala class to and from XML using the techniques shown in previous recipes in this chapter, i.e., a "manual conversion" process.
- Use a library like XStream to assist in the conversion process, writing converters as necessary.

This solution demonstrates both approaches.

The manual conversion process

The following code has a `toXml` method in the class to convert the fields in a class to XML, and a `fromXml` method in the object to convert an XML literal to the fields in a class, using a manual process. This example uses a `Stock` class, meant to hold data for an instance of a stock, such as `Stock("NFLX","Netflix",165.50)`:

```
class Stock(var symbol: String, var businessName: String, var price: Double) {

  // (a) convert Stock fields to XML
  def toXml = {
    <stock>
      <symbol>{symbol}</symbol>
      <businessName>{businessName}</businessName>
      <price>{price}</price>
    </stock>
  }

  override def toString =
    s"symbol: $symbol, businessName: $businessName, price: $price"

}

object Stock {

  // (b) convert XML to a Stock
  def fromXml(node: scala.xml.Node):Stock = {
    val symbol = (node \ "symbol").text
    val businessName = (node \ "businessName").text
    val price = (node \ "price").text.toDouble
    new Stock(symbol, businessName, price)
  }

}
```

The `toXml` method uses the variable substitution techniques shown in Recipe 1, and the `fromXml` method uses XPath search techniques demonstrated in Recipes 4 through 6. The `toXml` method is declared in the *class* because it needs to be unique to each instance, but the `fromXml` method is declared in the *object* because it's called as `Stock.fromXml(theXml)`, like a static method in Java.

The businessName field isn't necessary here, but is kept to make sure the stock symbols are understood.

These methods are demonstrated with the following driver class:

```
object TestToFromXml extends App {

  // (a) convert a Stock to its XML representation
  val aapl = new Stock("AAPL", "Apple", 600d)
  println(aapl.toXml)

  // (b) convert an XML representation to a Stock
  val googXml = <stock>
      <symbol>GOOG</symbol>
      <businessName>Google</businessName>
      <price>620.00</price>
    </stock>
   val goog = Stock.fromXml(googXml)
   println(goog)
}
```

Running the TestToFromXml object produces the following output:

```
<stock>
      <symbol>AAPL</symbol>
      <businessName>Apple</businessName>
      <price>600.0</price>
    </stock>

symbol: GOOG, businessName: Google, price: 620.0
```

This process is straightforward and uses techniques demonstrated in other recipes in this chapter. The only thing new in this recipe is using the toDouble method in this line of code:

```
val price = (node \ "price").text.toDouble
```

You can use toDouble and all the related to* methods, because the text method returns a String, which you can manipulate in all the usual ways.

Although writing code like this is a manual process, the code can easily be generated from the class specification or the database using a "CRUD generator."

Using XStream

Another solution to this problem is to use a library such as XStream, a popular Java library for serializing XML. Like other Java libraries, it works with Scala, though you'll need to provide a "converter" to handle special situations and collections.

For instance, you might start with the following simple class:

```
// a "first attempt" example (has a few problems)

import scala.collection.mutable.ArrayBuffer
import com.thoughtworks.xstream._
import com.thoughtworks.xstream.io.xml.DomDriver

case class Topping (name: String)

case class Pizza(crustSize: Int, crustType: String) {
  val toppings = ArrayBuffer[Topping]()
  def addTopping(t: Topping) { toppings += t }
}

object Test extends App {

  val p = Pizza(14, "Thin")
  p.addTopping(Topping("cheese"))
  p.addTopping(Topping("sausage"))

  val xstream = new XStream(new DomDriver)
  val xml = xstream.toXML(p)
  println(xml)

}
```

After including the XStream JAR in the project, running the `Test` object results in the following output, which has a few issues:

```
<Pizza>
  <crustSize>14</crustSize>
  <crustType>Thin</crustType>
  <toppings>
    <initialSize>16</initialSize>
    <array>
      <Topping>
        <name>cheese</name>
      </Topping>
      <Topping>
        <name>sausage</name>
      </Topping>
      <null/>
      <null/>
      <null/>
      ...
      (this goes on until 14 null fields are printed)
      ...
      <null/>
      <null/>
      <null/>
    </array>
    <size0>2</size0>
  </toppings>
</Pizza>
```

Issues in this output include:

- The `Pizza` and `Topping` tags are capitalized.
- All those `<null/>` tags generated from the `ArrayBuffer[Topping]`.
- The extra `<array>` and `<size>` tags.

To solve these problems, you need to create *aliases* to handle the capitalization problems, and write an XStream converter to handle the collection.

A converter class named `net.mixedbits.tools.XStreamConversions` provides a good start toward solving the collections problem. At the time of this writing, it currently has one "import" bug that's easily fixed, plus some deprecation issues.

After copying and pasting the `XStreamConversions` class into my project and fixing its import bug, the following program shows how to use it. In addition, the `alias` method calls show to convert the strings `Pizza` and `Topping` to lowercase:

```
// improved example

import scala.collection.mutable.ArrayBuffer
import com.thoughtworks.xstream._
import com.thoughtworks.xstream.io.xml.DomDriver
import com.thoughtworks.xstream.io.xml.StaxDriver

// added
import net.mixedbits.tools.XStreamConversions

case class Topping (name: String)

case class Pizza(crustSize: Int, crustType: String) {
  val toppings = ArrayBuffer[Topping]()
  def addTopping(t: Topping) { toppings += t }
}

object Test extends App {

  val p = Pizza(14, "Thin")
  p.addTopping(Topping("cheese"))
  p.addTopping(Topping("pepperoni"))

  // pass XStream into XStreamConversions
  val xstream = XStreamConversions(new XStream(new DomDriver()))
```

```
    // make Topping and Pizza lowercase
    xstream.alias("topping", classOf[Topping])
    xstream.alias("pizza", classOf[Pizza])

    val xml = xstream.toXML(p)
    println(xml)

  }
```

Running this code results in the following (improved) output:

```
<pizza>
  <crustSize>14</crustSize>
  <crustType>Thin</crustType>
  <toppings>
    <topping>
      <name>cheese</name>
    </topping>
    <topping>
      <name>pepperoni</name>
    </topping>
  </toppings>
</pizza>
```

A library like XStream can really save time and effort if you need to serialize and deserialize a large number of classes.

See Also

- The XStream library is "a simple library to serialize objects to XML and back again"
- Information about tweaking XStream output
- Writing an XStream converter

9. Loading XML from a URL

Problem

You want to load XML from a URL, such as accessing a SOAP web service or downloading an RSS feed.

Solution

For simple purposes, use the `load` method of the `scala.xml.XML` object directly:

```
val rss = scala.xml.XML.load("http://alvinalexander.com/rss.xml")
```

This is easily demonstrated in the Scala REPL:

```
scala> import scala.xml.XML
import scala.xml.XML

scala> val rss = XML.load("http://alvinalexander.com/rss.xml")
rss: scala.xml.Elem =
<rss xml:base="http://alvinalexander.com" version="2.0"↵
 xmlns:dc="http://purl.org/dc/elements/1.1/">
<channel>

(output goes on for a while ...)
```

Loading XML data from a URL is just like assigning an XML literal to a variable, with the resulting variable being of type `scala.xml.Elem`. As a result, you can operate on it as usual, such as determining the number of elements that match a search pattern, or extracting the text value of an element:

```
scala> (rss \\ "item").length
res0: Int = 10

scala> val firstTitle = (rss \\ "channel" \ "title").text
firstTitle: String = alvinalexander.com
```

This approach is simple, but doesn't let you control timeouts, a problem that will be solved in the Discussion.

Discussion

As with any attempt to access an Internet resource, the `load` method can throw exceptions, such as an `UnknownHostException`:

```
scala> val rss = XML.load("http://www.foo.bar/rss.xml")
java.net.UnknownHostException: www.foo.bar
```

If you dig into the Scala source code, you'll find that the URL is read with the `openStream` method of the `java.net.URL` class. The `openStream` method can throw an `IOException`, which you'll want to account for.

Using a timeout

Though the Solution shows a "simple" solution to the problem, in most production applications, you'll want to control the timeout, in case the resource you're trying to access is unavailable. One way to manually control the timeout limit while downloading a URL is to combine this recipe with Recipe 13.9, as shown in the following code:

```
import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.language.postfixOps
import scala.xml.XML

// for the 'pool' approach
import java.util.concurrent.Executors
```

```
object XmlTimeout extends App {

  val pool = Executors.newCachedThreadPool()
  implicit val ec = ExecutionContext.fromExecutorService(pool)

  try {
    val getXml = Future {
      XML.load("http://alvinalexander.com/rss.xml")
    }
    val rss = Await.result(getXml, 5 seconds)
    println(rss)
  } catch {
    case e: Exception => e.printStackTrace
  }

  pool.shutdown

}
```

The Scala `Future` lets you easily add a timeout to the operation. Scala Futures need something called an "execution context" to work, and the `pool` variable and the implicit value `ec` provide that for this application. The official Scala documentation describes this need like this:

> In order to execute callbacks and operations, Futures need something called an `ExecutionContext`, which is very similar to a `java.util.concurrent.Executor`. If you have an `ActorSystem` in scope, it will use its default dispatcher as the `ExecutionContext`, or you can use the factory methods provided by the `ExecutionContext` companion object to wrap `Executors` and `ExecutorServices`, or even create your own.

The following code is almost identical to the previous code, except that an `ActorSystem` is used to provide the `ExecutionContext`:

```
import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.language.postfixOps
import scala.xml.XML

object XmlTimeout extends App {

  implicit val system = ActorSystem("FutureSystem")

  try {
    val getXml = Future {
      XML.load("http://alvinalexander.com/rss.xml")
    }
    val rss = Await.result(getXml, 5 seconds)
    println(rss)
  } catch {
    case e: Exception => e.printStackTrace
  }

  system.shutdown

}
```

Either approach yields the desired result of downloading XML content with a timeout.

In these examples, I catch a generic `Exception`, but the code can actually throw many exceptions, including:

- `java.net.UnknownHostException`
- `java.util.concurrent.TimeoutException`
- `org.xml.sax.SAXParseException`

These are the most common exceptions I see, but because the specific exception types don't matter in my application, I just catch the top-level `Exception`.

If you prefer to use an HTTP client you're already comfortable with, you can use it to download the XML as a string, and then convert the string to an XML literal using the `scala.xml.XML.loadString` method.

See Also

- The `java.net.URL` class
- The official Scala documentation on Futures and Promises

10. Loading XML from a File

Problem

You need to open and read an XML data file for your application, such as loading configuration or application data.

Solution

Use the `loadFile` method of the `scala.xml.XML` object:

```
scala> import scala.xml.XML
import scala.xml.XML

scala> val xml = XML.loadFile("portfolio.xml")
xml: scala.xml.Elem =
<portfolio>
    <stocks>
```

```
      <stock>AAPL</stock>
      <stock>AMZN</stock>
      <stock>GOOG</stock>
    </stocks>
    <reits>
      <reit>Super REIT 1</reit>
    </reits>
</portfolio>
```

When the file is loaded, the resulting variable is of type is `scala.xml.Elem`, which is the same result you get when you assign an XML literal directly to a variable. Therefore, you can operate on it using the methods shown in other recipes, including the \ and \\ methods:

```
scala> xml \ "stocks" \ "stock"
res0: scala.xml.NodeSeq =
NodeSeq(<stock>AAPL</stock>, <stock>AMZN</stock>, <stock>GOOG</stock>)

scala> (xml \ "stocks" \ "stock").map(_.text)
res1: scala.collection.immutable.Seq[String] = List(AAPL, AMZN, GOOG)
```

Be aware that, as with any file-opening/reading code, `XML.loadFile` can throw exceptions:

```
scala> val xml = XML.loadFile("foo.xml")
java.io.FileNotFoundException: foo.xml (No such file or directory)
```

The process can also throw a `SaxParseException` like these:

```
org.xml.sax.SAXParseException: Premature end of file.
org.xml.sax.SAXParseException: Content is not allowed in prolog.
```

Therefore you'll want to wrap the code in a `try/catch` block, or throw the exception from your method.

Discussion

This approach is almost identical to Recipe 9, "Loading XML from a URL", but uses `XML.loadFile` instead of `XML.load`.

The `load` method is overloaded, and different instances let you load from a `URL`, `InputSource`, `Reader`, and `InputStream`. An additional `loadXML` method lets you load XML from an `InputSource` while also providing a parser. See the `scala.xml.XML` Scaladoc for more information.

See Also

The `scala.xml.XML` object includes a variety of "load" methods.

11. Saving XML to a File

Problem

You want to write XML data to a file, such as saving application data or configuration information to a file.

Solution

Use the `scala.xml.XML.save` method to write a Scala literal to a file. Given this XML literal:

```
// create an XML variable
val portfolio =
  <portfolio>
    <stocks>
      <stock>AAPL</stock>
      <stock>AMZN</stock>
      <stock>GOOG</stock>
    </stocks>
    <reits>
      <reit>Super REIT 1</reit>
    </reits>
  </portfolio>
```

write the literal to file with the `save` method:

```
// save the XML to a file
scala.xml.XML.save("portfolio.xml", portfolio)
```

This creates a plain-text file named *portfolio.xml* in the current directory, containing the XML literal shown.

As with any file-writing code, beware that `XML.save` can throw an exception:

```
scala> scala.xml.XML.save("/foo/bar/baz", portfolio)
java.io.FileNotFoundException: /foo/bar/baz (No such file or directory)
```

Additional save parameters

The `save` method lets you specify other parameters, including the encoding to use, whether or not to write an XML declaration, and whether or not to write a DOCTYPE declaration.

To write the data to a file with encoding information, use this approach:

```
XML.save("portfolio.xml", portfolio, "UTF-8", true, null)
```

This results in the following header being added to the file:

```
<?xml version='1.0' encoding='UTF-8'?>
```

To add a DOCTYPE to the file, first import the necessary classes:

```
import scala.xml.dtd.{DocType, PublicID}
```

Then create a `DocType` instance, and save the file with that instance:

```
val doctype = DocType("html",
  PublicID("-//W3C//DTD XHTML 1.0 Strict//EN",
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"),
  Nil)

XML.save("portfolio.xml", portfolio, "UTF-8", true, doctype)
```

With this configuration, the following DOCTYPE line is added to the output file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

The `scala.xml` package also includes a `Utility` object with a small collection of useful methods that let you sort node attributes, trim elements, remove comments, and more.

Pretty printing

If it's important that the XML can be easily read by a human being, it's best to use the `PrettyPrinter` class described in the next recipe to make sure the XML is "human readable." But because the `PrettyPrinter` returns a `String`, you'll need to write it to a file using another method. I have my own `FileUtils` class, so the process looks like this:

```
val portfolio =
  <portfolio>
    <stocks>
      <stock>AAPL</stock><stock>AMZN</stock><stock>GOOG</stock>
    </stocks>
    <reits><reit>Super REIT 1</reit></reits>
  </portfolio>

// 80 characters wide, 2 character indentation
val prettyPrinter = new scala.xml.PrettyPrinter(80, 2)
val prettyXml = prettyPrinter.format(portfolio)
FileUtils.save("portfolio.xml", prettyXml)
```

See Also

My Java file utilities and my Scala file utilities

12. Displaying XML in a Human-Readable Format (Pretty Printing)

Problem

You have some XML in a hard-to-read format and want to print it in a format that's easier to read, at least for humans.

Solution

Use the `scala.xml.PrettyPrinter` class. Imagine starting with a long, continuous string of XML:

```
scala> val x = <pizza><topping>cheese</topping><topping>sausage</topping></pizza>
x: scala.xml.Elem =
   <pizza><topping>cheese</topping><topping>sausage</topping></pizza>
```

The `toString` method prints the XML just as it was received:

```
scala> x.toString
res0: String = <pizza><topping>cheese</topping><topping>sausage</topping></pizza>
```

To print the XML in a more human-readable format, import the `scala.xml.PrettyPrinter` class, create a new instance of it, and format the XML as desired.

For instance, to improve the previous XML output, create a `PrettyPrinter` instance, setting the row width and indentation level as desired, in this case `80` and `4`, respectively:

```
scala> val p = new scala.xml.PrettyPrinter(80, 4)
p: scala.xml.PrettyPrinter = scala.xml.PrettyPrinter@4a3a08ea
```

Formatting the XML literal returns a `String`, formatted as specified:

```
scala> p.format(x)
res1: String =
<pizza>
    <topping>cheese</topping>
    <topping>sausage</topping>
</pizza>
```

As you might guess, the `PrettyPrinter` constructor looks like this:

```
PrettyPrinter(width: Int, step: Int)
```

The `width` is the maximum width of any row, and `step` is the indentation for each level of the XML.

There are other formatting methods available that let you specify namespace information, and a `StringBuffer` to append to. See the `PrettyPrinter` Scaladoc for more information.

See Also

The `PrettyPrinter` class

BONUS CHAPTER
Testing and Debugging

Introduction

When it comes to testing your code, the Scala community has come up with several great testing choices, including ScalaTest and specs2. Because ScalaTest supports both test-driven development (TDD) and behavior-driven development (BDD), is well documented and supported, and also has an Eclipse plug-in, it is covered in this chapter. But fear not, if you understand the BDD approach in ScalaTest, it's a simple matter to use another tool like specs2 for BDD testing. (A benefit of using specs2 is that it's the default testing tool in the Play Framework.)

In addition to covering ScalaTest, this chapter demonstrates several logging techniques, and also shows how to use JUnit in Scala. This can help you introduce Scala into an organization while letting everyone still use a comfortable tool.

1. Installing ScalaTest

Problem

You want to begin using ScalaTest to write TDD- or BDD-style unit tests for your Scala applications.

Solution

The easiest way to use Scala and ScalaTest in production projects is to use the Simple Build Tool (SBT) to manage your project, so the recipes in this chapter assume you'll be using SBT.

To use ScalaTest with SBT 0.10 or newer with Scala 2.10.0, just add a line like this to your *build.sbt* file:

```
libraryDependencies += "org.scalatest" %% "scalatest" % "1.9.1" % "test"
```

If you're more comfortable using JAR files, download the latest file from the ScalaTest download page.

Discussion

This recipe is intentionally short and simple, because your next step is to decide whether you want to use a TDD or BDD style of testing.

When deciding whether you want to write tests in a TDD or BDD style, it's important to know that ScalaTest provides classes to support a number of testing "styles." For instance, a `FunSuite` is meant to feel comfortable for developers coming to ScalaTest from JUnit/NUnit. It's used in the TDD-style tests in this chapter.

A `FlatSpec` offers "a nice first step for teams wanting to move from JUnit to a BDD style," whereas `FunSpec` is for teams coming from the Ruby RSpec tool. `FunSpec` is used in the BDD examples in this chapter.

Beyond these basic styles, there are still more. From the ScalaTest documentation:

> `WordSpec` is meant "for teams coming from specs or specs2."
>
> `FreeSpec` "gives absolute freedom on how specification text should be written."
>
> `PropSpec` is "for teams that want to write tests exclusively in terms of property checks."
>
> `FeatureSpec` "is primarily intended for acceptance testing, including facilitating the process of programmers working alongside non-programmers to define the acceptance requirements."

See the "Selecting a ScalaTest testing style" link in the See Also section for more details on choosing a style.

See Also

- Selecting a ScalaTest testing style
- My "Basic SBT Project with ScalaTest" project on GitHub
- ScalaTest download page
- The specs2 testing library

2. Writing TDD Tests with ScalaTest

Problem

You want to use TDD-style tests in your Scala applications and need to see examples of how to write them.

Solution

Have your test classes extend the ScalaTest `FunSuite`, and then write your tests. Because most tests involve a setup and teardown process, you'll usually also want to add in the `BeforeAndAfter` trait.

The following example demonstrates a simple set of TDD tests for a `Pizza` class:

```
package com.acme.pizza

import org.scalatest.FunSuite
import org.scalatest.BeforeAndAfter

class PizzaTests extends FunSuite with BeforeAndAfter {

  var pizza: Pizza = _

  before {
    pizza = new Pizza
  }

  test("new pizza has zero toppings") {
    assert(pizza.getToppings.size == 0)
  }

  test("adding one topping") {
    pizza.addTopping(Topping("green olives"))
    assert(pizza.getToppings.size === 1)
  }

  // mark that you want a test here in the future
  test ("test pizza pricing") (pending)

}
```

Assuming that you're using SBT with the default directory structure, this file should be named something like *PizzaTests.scala*, and should be placed under the *src/test/scala* directory structure, preferably in a directory named *src/test/scala/com/acme/pizza*. (The file can be placed anywhere under the *src/test/scala* directory, but I prefer to match the package and directory names.)

The accompanying *Pizza.scala* and *Topping.scala* files should be placed in the *src/main/scala/com/acme/pizza* directory. Here's the source code for the `Pizza` class:

```
package com.acme.pizza

import scala.collection.mutable.ArrayBuffer

class Pizza {

  private val toppings = new ArrayBuffer[Topping]
```

```
    def addTopping (t: Topping) { toppings += t}
    def removeTopping (t: Topping) { toppings -= t}
    def getToppings = toppings.toList

    def boom { throw new Exception("Boom!") }
  }
```

Here's the `Topping` class:

```
package com.acme.pizza

case class Topping(name: String)
```

With these classes in place, run the tests with the SBT `run test` command from your shell prompt, in your project's root directory:

```
$ sbt test
```

Your output will look like this:

```
[info] Compiling 1 Scala source to
       /Users/Al/ScalaTests/target/scala-2.10/classes...
[info] Compiling 1 Scala source to
       /Users/Al/Scala/ScalaTests/target/scala-2.10/test-classes...
[info] PizzaTests:
[info] - new pizza has zero toppings
[info] - adding one topping
[info] - test pizza pricing (pending)
[info] Passed: : Total 3, Failed 0, Errors 0, Passed 2, Skipped 1
```

Although there are other ways to run your tests, I'm a believer in using the tools of a language, so I recommend using SBT. See the ScalaTest guide to running your tests.

Discussion

The Scaladoc for the FunSuite class describes the class like this:

> A suite of tests in which each test is represented as a function value. The "Fun" in FunSuite stands for "function."

The `before` method lets you do any setup work that needs to be performed before each test is run. Similarly, the `after` method lets you perform any teardown work that should be performed after each test.

If you've used JUnit or other testing frameworks, the tests look familiar. Just describe your test with a unique string, and that string will be used in the printed output. (Make sure your strings are unique, or an exception will be thrown.)

The basic `assert` method is familiar, but it also has more power, as you'll see in other recipes.

The `pending` feature shown in the last test is very helpful:

```
test ("test pizza pricing") (pending)
```

This is a nice way of noting that you need to add a test in the future, but for one reason or another you're not adding that test right now. As you saw, this ends up in the printed output like this:

```
 [info] - test pizza pricing (pending)
```

See Also

- The Scaladoc for the FunSuite class
- Different ways to run your tests, including the Scala interpreter, Ant, Maven, Eclipse, IntelliJ IDEA, and more
- My "Basic SBT Project with ScalaTest" project on GitHub

3. Writing a First BDD Test with ScalaTest

Problem

You want to write your ScalaTest tests using a BDD style.

Solution

Extend the ScalaTest `FunSpec` trait, typically with the `BeforeAndAfter` trait. Then use the approach shown in the following `PizzaSpec` test class.

A series of tests begins with the `describe` method, with individual tests declared in `it` methods:

```
package com.acme.pizza

import org.scalatest.FunSpec
import org.scalatest.BeforeAndAfter

class PizzaSpec extends FunSpec with BeforeAndAfter {

  var pizza: Pizza = _

  before {
    pizza = new Pizza
  }

  describe("A Pizza") {

    it("should start with no toppings") {
      assert(pizza.getToppings.size == 0)
    }
```

```
      it("should allow addition of toppings") (pending)

      it("should allow removal of toppings") (pending)
    }

  }
```

Notice how `before`, `describe`, and `it` seem more like operators than methods. Scala gives developers the power to create their own domain specific languages (DSLs) like this.

This class should be placed in a file named *PizzaSpec.scala*, in a directory named *src/test/scala/com/acme/pizza*, under your root SBT folder. Assuming that you also have the `Pizza` and `Topping` classes installed as described in Recipe 2, "Writing TDD Tests with ScalaTest", you'll see the following output when you run the tests with `sbt test`:

```
[info] PizzaSpec:
[info] A Pizza
[info] - should start with no toppings
[info] - should allow addition of toppings (pending)
[info] - should allow removal of toppings (pending)
[info] Passed: : Total 1, Failed 0, Errors 0, Passed 1, Skipped 2
```

Discussion

As you can see, the output of a set of BDD-style tests reads like a software specification: "A Pizza should," followed by a series of statements about what the `Pizza` class should do. You'll see how this gets even better in subsequent recipes.

If you intentionally inject an error into your first test, setting the size to `1` (instead of `0`), the `sbt test` output will look like this instead:

```
[info] PizzaSpec:
[info] A Pizza
[info] - should start with no toppings *** FAILED ***
[info]   org.scalatest.exceptions.TestFailedException was thrown.
(PizzaSpec.scala:17)
[info] - should allow addition of toppings (pending)
[info] - should allow removal of toppings (pending)
[error] Failed: : Total 3, Failed 1, Errors 0, Passed 0, Skipped 2
[error] Failed tests:
[error]        com.acme.pizza.PizzaSpec
[error] {file:/Users/Al/Tests/ScalaTest1/}default-d6b943/test:test:
Tests unsuccessful
```

4. Adding Given/When/Then Behavior to BDD Tests

Problem

You want to make your tests more BDD-like by adding `Given/When/Then` behavior to them.

Solution

Mix the `GivenWhenThen` trait into your `FunSpec` BDD test, then add the `Given/When/Then` conditions, as shown in the following code:

```
package com.acme.pizza

import org.scalatest.FunSpec
import org.scalatest.BeforeAndAfter
import org.scalatest.GivenWhenThen

class PizzaSpec extends FunSpec with GivenWhenThen {

  var pizza: Pizza = _

  describe("A Pizza") {

    it ("should allow the addition of toppings") {
      Given("a new pizza")
      pizza = new Pizza

      When("a topping is added")
      pizza.addTopping(Topping("green olives"))

      Then("the topping count should be incremented")
      expectResult(1) {
        pizza.getToppings.size
      }

      And("the topping should be what was added")
      val t = pizza.getToppings(0)
      assert(t === new Topping("green olives"))
    }
  }

}
```

Assuming that you have the `Pizza` and `Topping` classes installed as described in Recipe 2, "Writing TDD Tests with ScalaTest", this class prints the following test output, which provides more detail to anyone reviewing the output:

```
[info] Compiling 1 Scala source to
/Users/Al/Projects/Scala/Tests/ScalaTest1/target/scala-2.10.0/test-classes...
[info] PizzaSpec:
```

```
[info] A Pizza
[info] - should allow the addition of toppings
[info]   + Given a new pizza
[info]   + When a topping is added
[info]   + Then the topping count should be incremented
[info]   + And the topping should be what was added
[info] Passed: : Total 2, Failed 0, Errors 0, Passed 2, Skipped 0
```

Discussion

Although Recipe 3, "Writing a First BDD Test with ScalaTest" got you started down the road of the BDD-style of testing, adding the `GivenWhenThen` trait gives you the additional features necessary to more fully implement the BDD style.

Dan North, the creator of the BDD style of testing, describes the `Given`/`When`/`Then` approach on his website. He shows that the basic pattern looks like this:

- Given some initial context (the givens)
- When an event occurs
- Then ensure some outcomes

A simple example of this might look like this:

- Given a new pizza
- When a topping is added
- Then the pizza should have one topping

A more complicated example might look like this:

- Given a new pizza
- When new toppings are added
- And the size is set
- And the crust type is set
- Then the pizza should have all of those toppings
- And the crust size should be correct
- And the crust type should be correct
- And the price of the pizza should be X

Adding more tests

As you can see from the output, the `Given` method prints `Given` before your string; `When` prints `When` before your string; and so on with `Then` and `And`.

As you add more BDD-style tests to your class:

```
it("Should start with no toppings") {
  Given("a new pizza")
  pizza = new Pizza
  Then("the topping count should be zero")
  assert(pizza.getToppings.size == 0)
}

it("Should allow removal of toppings") {
  Given("a new pizza with one topping")
  pizza = new Pizza
  pizza.addTopping(Topping("green olives"))

  When("the topping is removed")
  pizza.removeTopping(Topping("green olives"))

  Then("the topping count should be zero")
  expectResult(0) {
    pizza.getToppings.size
  }
}
```

you end up with test results that read like a software specification:

```
[info] Compiling 1 Scala source to ↵
/Users/Al/Projects/Scala/Tests/ScalaTest1/target/scala-2.10.0/test-classes...
[info] PizzaSpec:
[info] A Pizza
[info] - Should start with no toppings
[info]   + Given a new pizza
[info]   + Then the topping count should be zero
[info] - Should allow the addition of toppings
[info]   + Given a new pizza
[info]   + When a topping is added
[info]   + Then the topping count should be incremented
[info]   + And the topping should be what was added
[info] - Should allow removal of toppings
[info]   + Given a new pizza with one topping
[info]   + When the topping is removed
[info]   + Then the topping count should be zero
[info] Passed: : Total 3, Failed 0, Errors 0, Passed 3, Skipped 0
```

See Also

- A description of behavior-driven development on Wikipedia
- Dan North's original "Introducing BDD" article
- ScalaTest's `given/when/then` documentation

5. Adding More Tests and Test Suites

Problem

You want to add more tests and a main test suite to your application.

Solution

To add more tests to your project, just create new test classes. For instance, to add a set of TDD-style tests for the `Topping` class, just create a `ToppingTests` class in the *src/test/scala/com/acme/pizza* directory:

```
package com.acme.pizza

import org.scalatest.FunSuite
import org.scalatest.BeforeAndAfter

class ToppingTests extends FunSuite with BeforeAndAfter {

  // add tests here ...
  test("test topping quantity") (pending)

}
```

Assuming that you have the `Pizza`, `Topping`, and `PizzaTests` classes installed as described in Recipe 2, "Writing TDD Tests with ScalaTest", your test output will now look like:

```
[info] PizzaTests:
[info] - new pizza has zero toppings
[info] - adding one topping
[info] - adding and removing topping
[info] - catching an exception
[info] - test pizza pricing (pending)
[info] ToppingTests:
[info] - test topping quantity (pending)
```

To continue adding more tests, just add more test classes. Other recipes demonstrate how to control which tests are run.

See Also

ScalaTest does have a concept of "nested suites," but they aren't well documented at this time.

6. Printing Expected and Actual Values When a Test Fails

Problem

You'd like to have better output from your assert tests, output that shows the expected and actual values.

Solution

One approach is to use the === method (three = symbols) instead of ==. When an `assert` test fails, the === method output shows the two values from the test.

This can be demonstrated by putting an intentional error into a unit test while using ===. Here's a modified version of a TDD `test` method with an intentional error:

```
test("new pizza has zero toppings (version 2)") {
  // intentional error here; size should be 0
  assert(pizza.getToppings.size === 1)
}
```

Adding this test to the TDD test class in Recipe 2 results in the following output:

```
[info] PizzaTests:
[info] - new pizza has zero toppings (version 2) *** FAILED ***
[info]   0 did not equal 1 (PizzaTests.scala:18)
```

Using === informs you that "0 did not equal 1," and gives you the line number of the failure.

Another approach is to use `expectResult` instead of `assert`:

```
test("new pizza has zero toppings (version 3)") {
  expectResult(1) {
    pizza.getToppings.size
  }
}
```

Although it's a little more verbose, it gives more meaningful output, indicating what value was expected and what was actually found:

```
[info] PizzaTests:
[info] - new pizza has zero toppings (version 3) *** FAILED ***
[info]   Expected 1, but got 0 (PizzaTests.scala:19)
```

Both approaches are simple, and extremely helpful when debugging why a test failed. Use whichever you prefer, or both.

See Also

- Recipe 2, "Writing TDD Tests with ScalaTest"
- ScalaTest's "Using Assertions" documentation

7. Marking a Test as Pending

Problem

You want to note that a test needs to be created, but you're not ready to write it yet.

Solution

Instead of supplying the body of a test method, mark the test as *pending*. In the TDD style, create a pending test like this:

```
test ("should allow removal of toppings") (pending)
```

In the BDD style, create a pending test like this:

```
it("should allow removal of toppings") (pending)
```

When your tests are run, pending tests will be printed like this:

```
[info] - should allow addition of toppings (pending)
```

Discussion

When tests are run at the command line with SBT, pending lines are printed in a yellow(ish) color.

Marking tests as pending helps support the TDD style, and it's a convenient way of saying, "I need to write this test and implement the code behind it, but I haven't gotten there yet."

See Also

- Recipe 2, "Writing TDD Tests with ScalaTest"
- Recipe 3, "Writing a First BDD Test with ScalaTest"

8. Testing Expected Exceptions

Problem

You want to test a portion of your code that should throw an exception under certain conditions.

Solution

Use the `intercept` method to verify that the exception occurs. In the following example, the `boom` method will always throw an exception. The `intercept` method lets you catch that exception, so you can verify that this portion of the method works as desired:

```
test ("catching an exception") {
  val thrown = intercept[Exception] {
    pizza.boom
  }
  assert(thrown.getMessage === "Boom!")
}
```

Here's how this code works:

- If `boom` throws an exception, `intercept` will catch and return that exception. This lets you assign it to a variable like `thrown`.
- If `boom` completes normally, or throws an exception you weren't expecting, `intercept` will complete abruptly with a `TestFailedException`.

As shown, you catch the exception in a value named `thrown`, and then test the message from the exception inside an `assert` method.

Discussion

This example used `intercept` to catch the exception and `assert` to test the exception message, but this isn't necessary.

The following code shows that you don't have to catch the exception as an object and then test its message. Because `intercept` will end the test with a `TestFailedException` if your code doesn't throw an exception, your test can be as simple as this:

```
test ("catching an exception") {
  intercept[Exception] { pizza.boom }
}
```

If your code throws the exception, `intercept` catches it, and the test succeeds. (You expected it to throw an exception, and it did.)

Conversely, if your method doesn't throw an exception, `intercept` will make the test fail. The output looks like this:

```
[info] - catching an exception *** FAILED ***
[info]   Expected exception java.lang.Exception to be thrown,
         but no exception was thrown. (Test.scala:27)
```

Of course, you can also use a `try/catch` block to test that the exception occurs under the right situations, but `intercept` was created as a way to assist with this common testing pattern.

See Also

- Recipe 2, "Writing TDD Tests with ScalaTest"
- The ScalaTest "Using Assertions" page provides more information on using `intercept`.

9. Mark Your Tests with Tags So You Can Include or Exclude Them

Problem

You want a way to label your individual tests, so you can easily choose to include or exclude them when running your tests. For instance, you may want to tag long-running tests like database or web service tests, because they take a long time to run, and you don't want to run them all the time.

Solution

Create one or more custom tags, and then include those tags in your test specifications. When you run your tests, declare which tests you want to run or not run.

To demonstrate this, begin with the `Pizza` and `Topping` classes shown in Recipe 2 and the BDD test class defined in Recipe 3.

Next, add a new class in the *src/test/scala/com/acme/pizza* directory named *Tags.scala*, and add this content:

```
package com.acme.pizza

import org.scalatest.Tag

object DatabaseTest extends Tag("DatabaseTest")
```

This defines a tag named `DatabaseTest` that you can use in your tests.

Next, add the tag to your database tests. This is how you add a tag to an `it` BDD-style test:

```
// add the 'DatabaseTest' tag to the 'it' method
it("Should start with no toppings", DatabaseTest) {
  Given("a new pizza")
  pizza = new Pizza
  Then("the topping count should be zero")
  assert(pizza.getToppings.size == 0)
}
```

Now when you run your tests, you can specify which tests you want to include. To include tests with the `DatabaseTest` tag in the list of tests to be run, start SBT, and then issue the following `test-only` command from within the SBT shell:

```
$ sbt

sbt> test-only -- -n DatabaseTest
```

To exclude tests with the `DatabaseTest` tag from being run, use this approach:

```
sbt> test-only -- -l DatabaseTest
```

(The second example uses a lowercase letter "L.")

Notice that this uses a different version of the `it` method than was used in previous recipes. This example calls the two-argument version of the `it` method, whereas previous examples used the one-argument version. More on this shortly.

Discussion

Some software developers like to write long-running tests, such as database and web service tests, and others don't. The ability to tag tests in this manner lets everyone have their way. Database and web service tests can be run when desired and also excluded when desired. Just tag the database and web service tests with tags like `DatabaseTest` and `WsTest`, then include and exclude them when running your tests.

More generally, thinking beyond terms of long-running tests and short-running tests, this approach lets you test segments of your applications, in any way you want to define those segments. For instance, you can tag tests as Model, View, or Controller tests, or in any other way that makes sense for your application.

To add multiple tags to one BDD-style `it` test, add them as additional parameters:

```
it("Should start with no toppings", DatabaseTest, WsTest) {
  // test code here
}
```

To add tags to TDD-style tests, add the tags to your `test` method declarations:

```
test("new pizza has zero toppings", DatabaseTest, WsTest) {
  //assert(pizza.getToppings.size === 1)
```

```
    expectResult(0) {
      pizza.getToppings.size
    }
  }
```

There are additional ways to tag tests. For instance, the ScalaTest documentation shows how to use `taggedAs` with an `it` test in a class that extends `FlatSpec`:

```
it must "subtract correctly" taggedAs(SlowTest, DbTest) in {
  val diff = 4 - 1
  assert(diff === 3)
}
```

See the links in the See Also section for more details.

To run tests that have both tags `DatabaseTest` and `WsTest`, separate their names by blank spaces inside quotes with the `-n` flag:

```
sbt> test-only -- -n "DatabaseTest WsTest"
```

Or to exclude those tests, do the same thing with the `-l` flag:

```
sbt> test-only -- -l "DatabaseTest WsTest"
```

For information on controlling which tags to include or exclude using other tools, such as running ScalaTest from the command line or Ant, see the ScalaTest filtering tests doc, and the Ant task filtering doc.

See Also

- Recipe 2, "Writing TDD Tests with ScalaTest"
- Recipe 3, "Writing a First BDD Test with ScalaTest"
- Additional ways to tag your tests
- Running tests in SBT
- Specifying tags to include and exclude
- Using the ScalaTest Ant task
- The `FlatSpec` trait

10. Temporarily Disabling a Test

Problem

You want to temporarily disable one or more tests, presumably until you can get them working again.

Solution

When using BDD-style tests, change `it` method calls to `ignore`:

```
ignore ("A new pizza has zero toppings", DatabaseTest) {
  Given("a new pizza")
  pizza = new Pizza
  Then("the topping count should be zero")
  assert(pizza.getToppings.size == 0)
}
```

With TDD-style tests, change `test` method calls to `ignore`:

```
ignore("A new pizza has zero toppings", DatabaseTest) {
  //assert(pizza.getToppings.size === 1)
  expectResult(0) {
    pizza.getToppings.size
  }
}
```

When your tests are run, the tests you changed to `ignore` will result in output similar to the following:

```
[info] - A new pizza has zero toppings !!! IGNORED !!!
```

When run from the command line using SBT, this output is displayed in a yellow font.

Discussion

Many times when testing your code, you'll need to temporarily disable some tests until you can get them working again. Changing `it` and `test` to `ignore` is a simple way to change the tests so they'll be skipped over. The output is a hard-to-miss reminder that the tests are being ignored.

See Also

- Recipe 2
- Recipe 3, "Writing a First BDD Test with ScalaTest"
- More information on tagging your tests with `ignore`

11. Using Mock Objects with ScalaTest

Problem

You want to use a mock object framework in your ScalaTest tests, such as Mockito.

Solution

ScalaTest offers support for the following mock testing frameworks:

- ScalaMock
- EasyMock
- JMock
- Mockito

Because the support for each framework is similar, let's take a look at using Mockito.

Before starting, imagine that you have a login web service for your application, and rather than call the real web service during your tests, you just want to mock one up.

In your application, you have "login service" code in a file named *src/main/scala/tests/LoginService.scala*, which looks like this:

```
package tests

// a very simple User class
case class User(name: String)

// a LoginService must have a 'login' method
trait LoginService {
  def login(name: String, password: String): Option[User]
}

// the code for our real/live LoginService
class RealLoginService extends LoginService {
  // implementation here ...
}
```

Notice that there's a `LoginService` trait, and the `RealLoginService` implements that trait. By following this pattern, you can use Mockito to "mock" your `LoginService` trait in your unit tests.

The following code shows how to create and use a mock `LoginService` using Mockito and ScalaTest:

```
package tests

import org.scalatest.FunSuite
import org.scalatest.BeforeAndAfter
import org.scalatest.mock.MockitoSugar
import org.mockito.Mockito._

class LoginServiceTests extends FunSuite with BeforeAndAfter with MockitoSugar {

  test ("test login service") {
    // (1) init
    val service = mock[LoginService]

    // (2) setup: when someone logs in as "johndoe", the service should work;
    // when they try to log in as "joehacker", it should fail.
    when(service.login("johndoe", "secret")).thenReturn(Some(User("johndoe")))
    when(service.login("joehacker", "secret")).thenReturn(None)

    // (3) access the service
    val johndoe = service.login("johndoe", "secret")
    val joehacker = service.login("joehacker", "secret")

    // (4) verify the results
    assert(johndoe.get == User("johndoe"))
    assert(joehacker == None)

  }

}
```

Here's a quick description of the code:

1. In the "init" step, you create a mock version of the `LoginService`. Notice that this mocks a trait that doesn't have an implementation. Because you can do this, you don't have to write a separate `MockLoginService` class, which is nice. (You also don't have to access Test or Production versions of the "real" login service, which would slow down your unit tests.)
2. In the setup portion of the code, you work with Mockito to define how the mock `LoginService` should respond when it's given two sets of data. When "johndoe" logs in, the mock login service should return a `Some(User("johndoe"))` instance, and when "joehacker" attempts to log in, it should return a `None`.
3. In the "access" portion of the code, you write the code just like you would normally in your application. You call the `LoginService` instance with some data, and get objects in return.
4. In the "verify" portion of the code, you verify that you received the data you expected from your login service.

In more real-world tests, you'd do things slightly differently. First, you'd use a more robust `User` class. Second, you'd take the objects that are received from the `LoginService` and attempt to do more things with them. There isn't much value in testing a mock `LoginService` by itself, but because this mock service lets you test the next steps in your application, a mock service becomes a very useful thing.

This example is just intended to get you started in the right direction. The important part of the test is that you got a `User` object back from the `LoginService`, just as though you had called a real, live, production login service.

Steps to using Mockito

To use Mockito like this in a ScalaTest project, the first step is to include the Mockito JAR file in your project. Assuming you're using SBT, either add the Mockito JAR file to your project's *lib* directory, or add Mockito as a dependency in your *build.sbt* file. With ScalaTest 1.9.1, this is the correct line to add to your *build.sbt* file:

```
libraryDependencies += "org.mockito" % "mockito-all" % "1.8.4"
```

The next step is to add the necessary imports to your unit test classes. These were shown in the example, and are repeated here:

```
import org.scalatest.mock.MockitoSugar
import org.mockito.Mockito._
```

Next, mix the `MockitoSugar` trait into your test class:

```
class PizzaTests extends FunSuite with BeforeAndAfter with MockitoSugar {
```

Once you have these configuration steps out of the way, you can begin creating mock objects in your ScalaTest tests with Mockito.

See Also

There are many more ways to use Mockito and other mock object frameworks in your ScalaTest tests. These links will help you get started:

- The ScalaTest website shows many ScalaMock, EasyMock, JMock, and Mockito examples.
- The Mockito website has many more examples.

12. Java-Style Logging with SLF4J

Problem

You want to add Java-style logging to your Scala application, and you're comfortable with the Java SLF4J library.

Solution

Assuming you're using SBT, include the necessary SLF4J dependencies in your *build.sbt* file:

```
name := "SLF4JTest"

version := "1.0"

scalaVersion := "2.10.0"
```

```
libraryDependencies ++= Seq("org.slf4j" % "slf4j-api" % "1.7.5",
                            "org.slf4j" % "slf4j-simple" % "1.7.5")
```

Alternatively, download the SLF4J libraries and put them in your *lib* folder.

You can then use SLF4J as you would with Java, as shown in this example, creating a `logger` instance, and then using it in your code:

```
import org.slf4j.Logger
import org.slf4j.LoggerFactory

class Pizza {
  val logger = LoggerFactory.getLogger(classOf[Pizza])
  logger.info("Hello from the Pizza class")
}

object Main extends App {
  val p = new Pizza
}
```

Running that file with `sbt run` prints several lines of output, including this line from `logger.info`:

```
[run-main] INFO Pizza - Hello from the Pizza class
```

Discussion

SLF4J is a popular Java logging solution, and as you can see from this example, it can be used the same way in a Scala application.

Although this is a nice approach, it can be improved by making it a little more "Scala like." To that end, see the next recipe, "Scala-Style Logging with Grizzled-SLF4J", which demonstrates how to use the Grizzled-SLF4J library to improve on this approach.

See Also

The SLF4J library

13. Scala-Style Logging with Grizzled-SLF4J

Problem

You want to add logging to an application in a more Scala-specific way than simply using SLF4J.

Solution

Grizzled-SLF4J is a thin wrapper around SLF4J that gives you logging in a more Scala-like way.

To get started with Grizzled-SLF4J, create a simple SBT project as shown in Recipe 18.1. Then edit your *build.sbt* file so it has the dependencies you'll need:

```
name := "Grizzled"

version := "1.0"

scalaVersion := "2.10.0"

libraryDependencies ++= Seq("org.slf4j" % "slf4j-api" % "1.7.5",
                            "org.slf4j" % "slf4j-simple" % "1.7.5",
                            "org.clapper" %% "grizzled-slf4j" % "1.0.1")
```

Then create a simple test class to use Grizzled-SLF4J. Put the following code in a file named *Main.scala* in the root directory of your SBT project:

```
import grizzled.slf4j.Logger

object Main extends App {

  // create a logger manually using one of these approaches
  //val logger = Logger("com.alvinalexander.test.Main")
  //val logger = Logger(classOf[Main])
  val logger = Logger[this.type]

  logger.info("Hello, world")

}
```

As you can see from that example, you can import the Grizzled-SLF4J `Logger` class, then manually create a `Logger` instance in several different ways in a class or object. Running this object with `sbt run` prints the following output to STDOUT:

```
[run-main] INFO Main$ - Hello, world
```

As a second approach, instead of manually creating a `logger` instance, you can mix in the Grizzled-SLF4J `Logging` trait, which creates the `logger` instance for you:

```
import grizzled.slf4j.Logging

object Main extends App with Logging {

  logger.info("Hello, world")

}
```

Running your object with this approach yields the same output as before:

```
[run-main] INFO Main$ - Hello, world
```

Discussion

Grizzled-SLF4J is a simple Scala wrapper around the well-known SLF4J library, has good documentation, and has recently been updated. Other libraries take a similar approach, but Grizzled-SLF4J currently has the best documentation and most recent updates.

A nice feature of Grizzled-SLF4J is that parameters to method calls like `log.info()` and `log.debug()` are *call-by-name* parameters. For instance, the Grizzled-SLF4J documentation shows that the `debug` method is defined like this:

```
@inline final def debug(message: => Any) =
    if (debugIsEnabled) log(message)
```

The documentation states:

> "Thus, `debug()` isn't a method taking a string; instead, it's a method taking a function that returns a string ... However, because `message` is a function that returns a string, it isn't evaluated until it is called—which is after the test that determines whether it should be logged."

Because `message` is a call-by-name parameter, it isn't evaluated until after the `debugIsEnabled` check, and if `debugIsEnabled` is `false`, it won't be evaluated at all. This is another benefit of using Grizzled-SLF4J.

See Also

- A discussion of how Grizzled-SLF4J uses the call-by-name approach
- The Grizzled-SLF4J library
- The SLF4J library

14. Using JUnit with Scala

Problem

You want to test your Scala code using JUnit.

Solution

Include the JUnit library in your project, and use it in the same way you've used it in Java projects, with a few minor changes.

Assuming you're using SBT on your project, include JUnit into the project by adding this dependency line to your *build.sbt* file:

```
libraryDependencies += "com.novocode" % "junit-interface" % "0.8" % ↵
"test->default"
```

According to the SBT testing documentation, "support for JUnit is provided by junit-interface," so you add it as a dependency here.

Next, if you're also using Eclipse, generate your Eclipse project files:

```
$ sbt eclipse
```

Then start Eclipse and import your project as usual.

In your project, create these simple model classes in a file named *Pizza.scala* under the *src/main/scala* directory:

```
package com.acme.pizza

import scala.collection.mutable.ArrayBuffer

case class Topping(name: String)

class Pizza {

  private var toppings = new ArrayBuffer[Topping]

  def addTopping (t: Topping) { toppings += t}
  def removeTopping (t: Topping) { toppings -= t}
  def getToppings = toppings.toList

}
```

(Any directory under *src/main/scala* is fine. Unlike Java, Scala package names don't have to match the directory structure.)

Next, create a JUnit test class as usual. Put the following code in a file named *PizzaTests.scala* under the *src/test/scala* directory:

```
package com.acme.pizza

import org.junit.Test
import junit.framework.TestCase
import org.junit.Assert._

class PizzaTests extends TestCase {

  var pizza: Pizza = _

  override def setUp {
    pizza = new Pizza
  }
  def testOneTopping {
    pizza.addTopping(Topping("green olives"))
    assertEquals(pizza.getToppings.size, 1)
  }

  def testAddingAndRemovingToppings {
    pizza.addTopping(Topping("green olives"))
    pizza.removeTopping(Topping("green olives"))
    assertEquals(pizza.getToppings.size, 0)
  }

}
```

If you're using Eclipse, run the tests from the Eclipse menu system. Choose Run → Run As → JUnit Test, and you should see your tests run successfully.

To run the tests using SBT, run the `sbt test` command from your operating system's command line, in the root directory of your SBT project:

```
$ sbt test
```

The output shows that the tests passed:

```
[info] Passed: : Total 2, Failed 0, Errors 0, Passed 2, Skipped 0
[success] Total time: 1 s
```

Discussion

Beyond running a single, simple JUnit test class, you can also create additional test classes, and then a `TestSuite`. To demonstrate this, first create a new test class named *ToppingTests.scala* in the *src/test/scala* directory:

```
package com.acme.pizza

import org.junit.Test
import junit.framework.TestCase
import org.junit.Assert._

class ToppingTests {

  @Test
  def foo {
    val t1 = Topping("cheese")
    val t2 = Topping("cheese")
    assertEquals(t1, t2)
  }

}
```

Then create a `TestSuite` class named `PizzaTestSuite` and save it in a file named *PizzaTestSuite.scala* in the *src/test/scala* directory:

```
package com.acme.pizza

import org.junit.runner.RunWith
import org.junit.runners.Suite

@RunWith(classOf[Suite])
@Suite.SuiteClasses(Array(classOf[PizzaTests], classOf[ToppingTests]))
class PizzaTestSuite
```

If you're using Eclipse, with this class in the editor, choose Run → Run As → JUnit Test, and you should see both your `PizzaTests` and `ToppingTests` run successfully.

If you're using SBT, just run `sbt test` again.

As you've seen in these examples, it's simple to use Scala with JUnit. The following lines show the major differences between using Scala instead of Java:

```
@RunWith(classOf[Suite])
@Suite.SuiteClasses(Array(classOf[PizzaTests], classOf[ToppingTests]))
```

In the first line, `classOf[Suite]` is used because Scala uses this syntax instead of `Suite.class`, which is the syntax Java uses. You do the same thing in the second line, and also replace the Java curly brackets array syntax with an `Array` declaration. Everything else is a straightforward port from Java to Scala.

JUnit has some advantages compared to ScalaTest or specs2. One advantage is that test execution is generally faster with JUnit. Also, you can use JUnit if you want to, and import the ScalaTest or the specs2 matchers inside your test cases if you want more expressivity for your assertions.

That being said, if you're going to code in Scala, I recommend using ScalaTest or specs2 for testing. As a friend likes to say, when we moved from C to Java, we didn't attempt to bring `make` along with us; we switched to Ant or Maven, so I'm a firm believer in using native Scala tools like SBT, ScalaTest, and specs2.

See Also

The SBT "testing" documentation

BONUS CHAPTER

The Play Framework

Introduction

There are several good frameworks for developing web applications in Scala, including the Lift Framework and Play Framework (Play). This chapter provides a collection of recipes for Play.

If you've used other web frameworks like Ruby on Rails or CakePHP, the Play approach will seem familiar. Like those frameworks, Play uses "convention over configuration" as much as possible, and even the directory layout is similar.

Play has many great features, including support for popular web development technologies like CoffeeScript and LESS. A really terrific feature is that Play uses templates, and those templates compile to normal Scala functions. As a result, it's easy to accomplish many tasks that are difficult in other frameworks, including creating one or more "master" templates to provide a common look and feel across a website, and the ability to easily include one template into another as a reusable widget.

Off the shelf, Play includes a database library named Anorm, which stands for "Anorm is Not an Object Relational Mapper." As its name implies, Anorm lets you write your data access objects (DAOs) using plain SQL. It's straightforward to use, and provides a DSL for its tasks. However, if Anorm isn't your cup of tea, Play makes it easy to plug in other database access technologies, such as Hibernate, JPA, and others.

Finally, you can deploy your Play application in several different ways, including the `dist` method, which lets you package your applications and all dependencies into a ZIP file, and only requires a JVM on the production server. This lets you easily deploy Play applications to application server environments from Amazon, Google, Heroku, and many others.

1. Creating a "Hello, World" Project

Problem

You want to create a new Play project, and understand the basics of the Play architecture.

Solution

Download and install the Play Framework distribution per the instructions on the Play website. Once it's installed, move to a directory where you normally create your projects, and then issue the `play new` command followed by your project name to create a new project. When prompted, choose the option to create a Scala application.

For instance, I keep my projects in a directory named */Users/Al/Projects*, so I follow these steps to create a new project named `Hello`:

```
$ cd /Users/Al/Projects

$ play new Hello

       _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|            |__/

play! 2.1.1

The new application will be created in /Users/Al/Projects/Hello

What is the application name? [Hello]
> [Enter]

Which template do you want to use for this new application?

  1 - Create a simple Scala application
  2 - Create a simple Java application

> 1
OK, application Hello is created.

Have fun!
```

With the project directory created, `cd` into that directory, and then start the Play command-line tool:

```
$ cd Hello

$ play
```

```
[info] Loading global plugins from /Users/Al/.sbt/plugins
[info] Updating {file:/Users/Al/.sbt/plugins/}default-6315be...
[info] Resolving org.scala-sbt#precompiled-2_10_0;0.12.2 ...
[info] downloading
[info] Done updating.
[info] Loading project definition from Hello/project
[info] Set current project to Hello
       _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|            |__/

play! 2.1.1

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[Hello] $ _
```

This is the Play command-line prompt. From here, you can start the Play server. Just type `run` to start the server on port `9000`, but if that port is already taken by another application (such as Eclipse), specify a different port for it to run on:

```
[Hello] $ run 8080

--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on /0.0.0.0:8080

(Server started, use Ctrl+D to stop and go back to the console...)
```

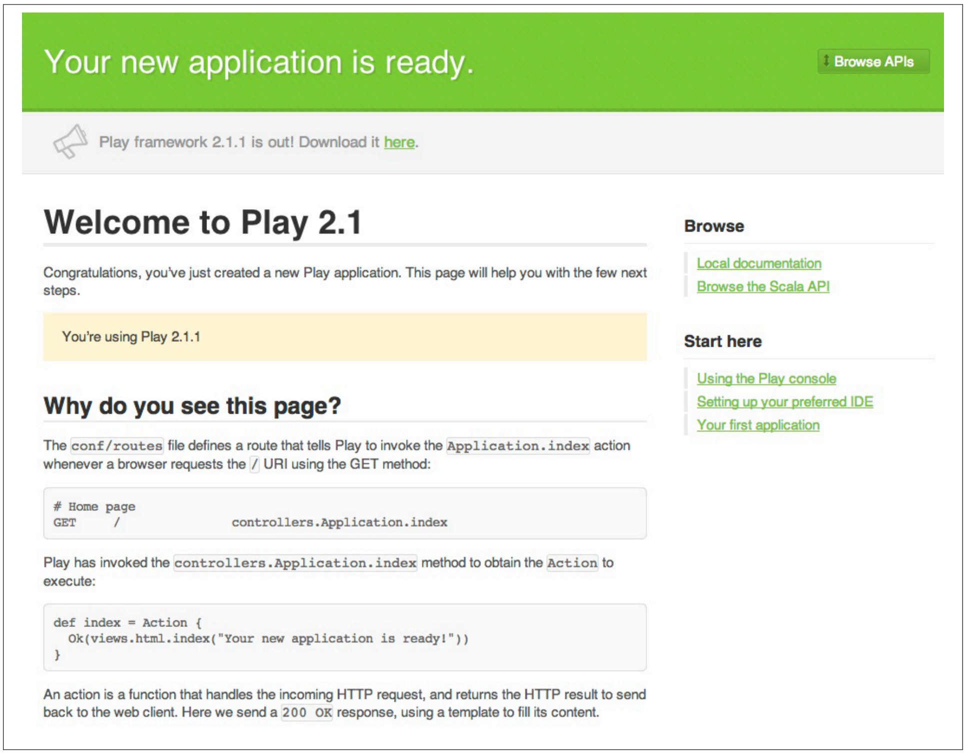
When you see the "Server started" message, the Play server is running, and you can now access your application from a browser. Because I started the server on port `8080`, I access the *http://localhost:8080/* URL in my browser, and after a few moments, I see the result shown in Figure 1.

There was probably a slight pause before this content was displayed in your browser. Looking back at the Play console, you'll see why. Play automatically compiled the source code for your application when you accessed that URL:

```
(Server started, use Ctrl+D to stop and go back to the console...)

[info] Compiling 5 Scala sources and 1 Java source to
       target/scala-2.10/classes...
[info] play - Application started (Dev)
```

Congratulations, your first Play application is now up and running.


Your new application is ready.

Play framework 2.1.1 is out! Download it here.

Welcome to Play 2.1

Congratulations, you've just created a new Play application. This page will help you with the few next steps.

You're using Play 2.1.1

Why do you see this page?

The `conf/routes` file defines a route that tells Play to invoke the `Application.index` action whenever a browser requests the `/` URI using the GET method:

```
# Home page
GET     /                controllers.Application.index
```

Play has invoked the `controllers.Application.index` method to obtain the `Action` to execute:

```
def index = Action {
  Ok(views.html.index("Your new application is ready!"))
}
```

An action is a function that handles the incoming HTTP request, and returns the HTTP result to send back to the web client. Here we send a `200 OK` response, using a template to fill its content.

Figure 1. The Play "Welcome" message

If you prefer to start Play on port `8080` from your operating system command line (rather than the Play shell), use this command:

```
$ play "run 8080"
```

If you want to run in debug mode using port 8080, use this command:

```
$ play debug "run 8080"
```

This starts a JPDA debug port you can connect to with a Java debugger.

Discussion

A Play application consists of the following components:

- *Controllers* that are placed in an *app/controllers* folder.
- *Templates* that are placed in an *app/views* folder.

- *Models* in an *app/models* folder. (This folder is not automatically created.)
- A *mapping* of application URIs to controller actions in the *conf/routes* file.

Other important files include:

- Application configuration information in the *conf/application.conf* file.
- Database scripts in the *conf/evolutions* folder. (Optional.)
- Frontend, design assets in the *public/images*, *public/javascripts*, and *public/stylesheets* folders.

If you're an Eclipse user, you can load the `Hello` project into Eclipse. If your Hello application is still running, press Ctrl-D at the Play command line. This brings you back to Play's `[Hello]` prompt:

```
[Hello] $
```

Type `eclipse` to have Play generate the *.project* and *.classpath* files for Eclipse:

```
[Hello] $ eclipse
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] Hello
```

Now import your project into Eclipse. From the Eclipse menu, select File → Import... → Existing Projects Into Workplace, click Next, and then navigate your filesystem and choose the `Hello` project you just created. When you open the project folders, your view should look like Figure 2.

To examine the files in the project, first look at the *conf/routes* file. In Play 2.1.1, this file contains the following default contents:

```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# Home page
GET    /                    controllers.Application.index

# Map static resources from the /public folder to the /assets URL path
GET    /assets/*file        controllers.Assets.at(path="/public", file)
```

For the purposes of understanding how the welcome page was displayed, this is the important line in that file:

```
GET   /     controllers.Application.index
```

This line can be read as, "When the HTTP GET method is called on the / URI, call the `index` method defined in the `Application` object in the *controllers* package." If you've used other frameworks like Ruby on Rails and CakePHP, you've seen this sort of thing

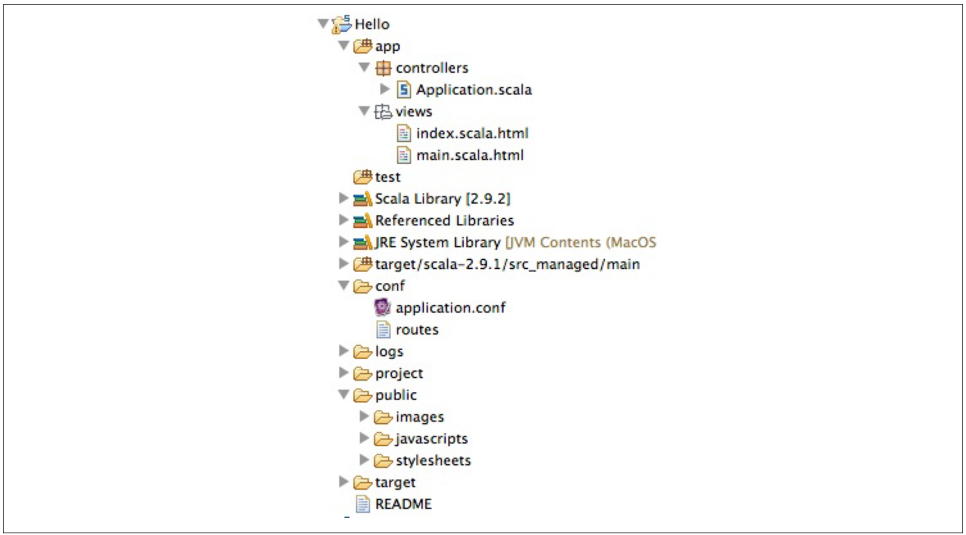

```
Hello
  app
    controllers
      Application.scala
    views
      index.scala.html
      main.scala.html
  test
  Scala Library [2.9.2]
  Referenced Libraries
  JRE System Library [JVM Contents (MacOS
  target/scala-2.9.1/src_managed/main
  conf
    application.conf
    routes
  logs
  project
  public
    images
    javascripts
    stylesheets
  target
  README
```

Figure 2. The directory structure of a new Play project, shown in Eclipse

before. It binds a specific HTTP method (such as GET or POST) and a URI to a method in an object.

Next, open the *app/controllers/Application.scala* file and look at the `index` method:

```
package controllers

import play.api._
import play.api.mvc._

object Application extends Controller {

  def index = Action {
    Ok(views.html.index("Your new application is ready."))
  }

}
```

This is a normal Scala source code file, with one method named `index`. This method implements a Play `Action` by calling a method named `Ok`, and passing in the content shown. The code `views.html.index` is the Play way of referring to the *views/index.scala.html* template file. A terrific thing about the Play architecture is that Play templates are compiled to Scala functions, so what you're actually seeing in this code is a normal function call:

```
views.html.index("Your new application is ready.")
```

This code essentially calls a function named `index` and passes it the string, "Your new application is ready."

Knowing that a template compiles to a normal Scala function, open the *app/views/index.scala.html* template file. You'll see the following contents:

```
@(message: String)

@main("Welcome to Play 2.1") {

    @play20.welcome(message)

}
```

Notice the first line of code:

```
@(message: String)
```

If you think of the template as a function, this is the parameter list of the function. This declares that the function takes one parameter, a `String` with the variable name `message`.

The `@` symbol in this file is a special character in a Play template file. It indicates that what follows is a Scala expression. For instance, in the line of code shown, the `@` character precedes the function parameter list. In the third line of code, the `@` character precedes a call to a function named `main`. Notice in that line of code, the string "Welcome to Play 2.1" is passed to the `main` method.

As you might have guessed, though `main` looks like a function, it's also a template file. When the code calls `main`, it actually invokes the *app/views/main.scala.html* template. Here's the source code for *main.scala.html*:

```
@(title: String)(content: Html)

<!DOCTYPE html>

<html>
  <head>
    <title>@title</title>
    <link rel="stylesheet" media="screen"
          href="@routes.Assets.at("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.at("images/favicon.png")">
    <script src="@routes.Assets.at("javascripts/jquery-1.9.0.min.js")"
            type="text/javascript"></script>
  </head>
  <body>
    @content
  </body>
</html>
```

This file is the default "wrapper" template file for the project. If every other template file calls `main` in the same way the *index.scala.html* file calls `main`, you can be assured that those templates will be wrapped with this same HTML, and as a result, all of your pages will have the same look and feel.

Notice the first line of this file:

```
@(title: String)(content: Html)
```

This template file (again, a function) takes two parameter lists. The first parameter list contains a variable named `title` of type `String`. It's used in the template between the `<title>` tags.

The variable in the second parameter list is named `content` and is of type `Html`. Near the end of this file, you'll see that this variable is emitted inside of `<body>` tags like this:

```
<body>
  @content
</body>
```

When you access the `/` URI in your browser, this is where the content from the *index.scala.html* file is emitted. Looking back at the `main` method call in the *index.scala.html* file, you can see how this works:

```
@main("Welcome to Play 2.1") {

    @play20.welcome(message)

}
```

The string "Welcome to Play 2.1" is passed as the first parameter to the `main` function (where it becomes the `title` parameter). The rest of the template is created as a block inside curly braces, and that block is passed in the second parameter list to the `main` function. Because the `main` function is actually the template *main.scala.html*, this block becomes the variable named `content` in that template, and the block is emitted inside the `<body>` tags in that file.

The following line of code in the *index.scala.html* file is what generates all the content you see in the browser:

```
@play20.welcome(message)
```

You can delete this code and replace it with something else, for instance, the usual "Hello, world" greeting. While you're at it, add a comment to the code using Play's `@* ... *@` comment syntax:

```
@(message: String)

@* this is a comment *@
@* ignoring the 'message' that's passed in *@
@main("Welcome to Play 2.1") {
    <h1>Hello, world</h1>

}
```

Save this file, then go back to the Play console and restart the server, if necessary:

```
[Hello] $ run 8080
```

Now refresh your browser and after a few moments, you'll see the "Hello, world" message. Congratulations, you've now seen all the basics of the Play Framework.

The Play console

Under the covers, the Play console is a normal SBT console, so you can run the usual SBT commands, such as `doc`, to generate Scaladoc:

```
[Stocks] $ doc
```

If you think there's a problem with SBT (the cache is corrupted), use the `clean` command:

```
[Stocks] $ clean
```

You can also run the Play `clean-all` command from your operating system command line:

```
$ play clean-all
[info] Done!
```

The Play `console` command opens a REPL session with your code loaded, so you can test it. To demonstrate this, the examples in this chapter use a `Stock` class in the *models* package, and you can create an instance of a `Stock` from the console:

```
[Stocks] $ console
[info] Updating
more output here ...
[info] Compiling 12 Scala sources and 1 Java source to
       target/scala-2.10/classes...
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.10.0.

scala> import models._
import models._

scala> val s = Stock(0, "NFLX", Some("Netflix"))
s: models.Stock = Stock(0,NFLX,Some(Netflix))
```

You can access other project classes and objects in the same way.

When you're finished, press Ctrl-D to exit the `scala>` prompt and return to the Play console.

As shown in the examples in this chapter, use the `run` command to run your application in development mode. However, don't use this command to run an application in production. The Play documentation states that for each request that's made when using the `run` command, a complete check is handled by SBT—definitely not something you want in production. Recipe 16 shows how to deploy a Play Framework project to production.

Summary

Here's a quick summary of what was demonstrated.

A Play application consists of the following components:

- The *conf/routes* file maps URIs and HTTP methods to controller methods.
- Controller classes are placed in the *app/controllers* folder.
- Controllers have methods, like the `index` method. These methods typically perform some business logic and then display a template, passing data to the template as needed.
- Templates are placed in the *app/views* folder.
- Template files are compiled to functions and can be called like functions.
- An application will usually have one or more master or "wrapper" template files, like the *main.scala.html* template that's automatically created for you. Other template files call these master template files so your application will have a consistent look and feel.
- Although this example didn't show it, model files (like a `Person`, `User`, `Order`, etc.) are placed in the *app/models* folder.

Other important files include:

- Application configuration information in the *conf/application.conf* file. This includes information on how to access a database.
- Database scripts in the *conf/evolutions* folder. (Optional.)
- Frontend, design assets in the *public/images*, *public/javascripts*, and *public/stylesheets* folders. The *main.scala.html* demonstrates the syntax for referring to these files.

See Also

- The Play Console page has more information on console commands.
- Starting your application in production mode.

2. Adding a Route, Model, and Controller Method to a Play Application

Problem

You need to see how to add a new route, controller method, and model to create new content at a new URI in a Play application.

Solution

Follow these steps to create new content at a new URI:

1. Create a new route in the *conf/routes* file.
2. The new route points to a controller method, so create that controller method.
3. The controller method typically forwards to a new template, so create that template.
4. The controller method may also require a model class, so create that class as needed.

To demonstrate this process, you'll add on to the code created in Recipe 1. You'll create new code to handle a GET request at */people*. This URI will return a list of `Person` instances in an HTML format.

Create a new route

To begin, you know you want to handle a new URI at */people*, so add a new route to the *conf/routes* file. This will be an HTTP GET request, so map the URI by adding this line to the end of the file:

```
GET   /people   controllers.Users.people
```

This can be read as, "When a GET request is made at the */people* URI, invoke the `people` method of the `Users` class in the *controllers* package."

Create a new controller method

Next, create the `Users` object in a *Users.scala* file in the *app/controllers* directory. Add the following code to that class:

```
package controllers

import play.api._
import play.api.mvc._
import models.Person

object Users extends Controller {

  def people = Action {
    val people = Person.getAll
    Ok(views.html.people(people))
  }

}
```

When the `people` method in this class is invoked, it gets a `List` of `Person` instances from the `Person` object by calling the `getAll` method. It then passes that `List` to a new template named *people.scala.html*. Neither the `Person` class (and object) nor the template exist yet, so you'll create them next.

Create a new model

To create the `Person` code, first create a *models* directory under the *app* directory. It should be at the same level as the *controllers* and *views* folders. Then create a new file named *Person.scala* under the *models* directory. Place these contents into that file:

```
package models

case class Person(name: String)

object Person {

  def getAll = List(Person("Al"), Person("Darren"), Person("Rich"))

}
```

This file consists of a case class named `Person`, and its companion object with a method named `getAll`. Although this example is simple, if you can imagine that the companion object is accessing a database, this approach follows the database access pattern shown in Play's Anorm documentation (and that I use personally): the model class and companion object are created in the same file, and the companion object has the code that accesses the database; i.e., it is the data access object (DAO).

Create a new template

Next, create a *people.scala.html* template file in the *views* directory. Add the following code to this file:

```
@(people: List[Person])

@main("Our List of People") {
  <h1>People</h1>

  <ul>
    @people.map { person =>
      <li>
        @person.name
      </li>
    }
  </ul>

}
```

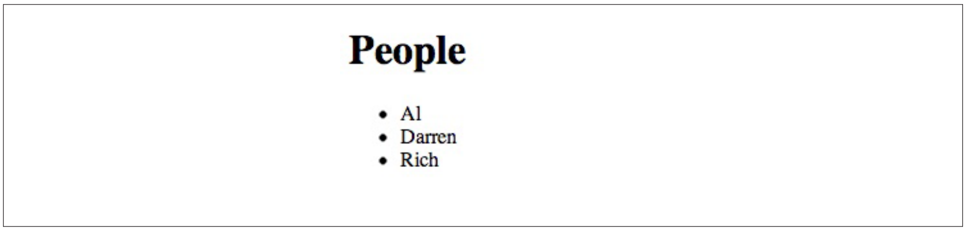
This template takes one parameter, a `List[Person]` named `people`. As shown in Recipe 1, the `@main` line invokes the *main.scala.html* wrapper template, passing it the string "Our List of People" as its first parameter list. It then passes it the block of code shown as its second parameter list. (Technically, these are two separate parameter lists, but you can think of them as two parameters, if you prefer.) Because `people` is a `List[Person]`, the `map` method is used to print the names from the `Person` instances in an unordered list using `<ul>` and `<li>`.

With all the code in place, go back to the Play console and restart the server, if necessary:

```
[Hello] $ run 8080
```

(If the server is already running, there's no need to restart it; another great Play feature.)

Then go to your browser and enter the URL *http://localhost:8080/people*, and you should see the result shown in Figure 3.


People

- Al
- Darren
- Rich

Figure 3. The output from the people.scala.html template displayed at the /people URI

Looking back at the Play console, you should see some output like this:

```
 [Hello] $ run 8080

[info] play - Listening for HTTP on port 8080...

(Server started, use Ctrl+D to stop and go back to the console...)

[info] Compiling 5 Scala sources and 1 Java source to
       target/scala-2.10.0/classes...
[info] play - Application started (Dev)
```

If your page wasn't displayed, and you don't see this output, press Ctrl-D to get back to the Play prompt, and restart the server with the `run` command:

```
[Hello] $ run 8080
```

(I haven't seen this happen too often, but if Play fails to recompile your application, this solves the problem.)

Discussion

As demonstrated, creating content at a new URI is typically a four-step process. The example followed these steps to emit the new content at the */people* URI:

1. You created a new route for the */people* URI in the *conf/routes* file.
2. That new route mapped to a method named `people` in a controller named `Users`, so you created that controller and method.
3. The controller method forwards to the *people.scala.html* template file, so you created that template.
4. The controller got its information from the `Person` model, so you created that class and its companion object.

There are a few other points worth mentioning. First, you didn't have to create a new controller; you could have just added the `people` method to the existing `Application` controller. However, this approach is beneficial because it shows the steps required to add a new controller, and it's representative of what you'd do in the real world.

Importing members into templates

Also, you may have noticed that you didn't have to import the `Person` class into the *people.scala.html* template file. Template files automatically import the `controllers._` and `models._` members, so an import statement isn't needed. You'll see in future recipes how to work with imports, but as a quick preview, all you have to do is add the import statements after the first line of the template:

```
@(people: List[Person])

import com.foo.Foo
import org.bar.Bar

<!-- more code here ... -->
```

3. Using Multiple Template Wrappers

Problem

The previous recipes demonstrated how to use one master (or wrapper) template that you can use to wrap all your template files to give your application a consistent look and feel, but in a production application you want to use multiple templates. For instance, you may want to have one template for the home page, one for a shopping cart area of a website, another for a blog, etc.

Solution

The Play Framework template approach makes this very easy. Just create a new wrapper template for each area of the website, and then call the desired wrapper template from within your other templates, just like the main template is called in Recipes 1 and 2.

For instance, create three wrapper template files with the following names in the *app/views* folder:

- *main.scala.html*
- *cart.scala.html*
- *blog.scala.html*

For the purposes of this recipe, you can create the last two files by copying and pasting the *main.scala.html* template file that Play generates for you. Then modify each template file slightly so you'll be able to see the difference between them in a browser. For instance, add a different `<h1>` tag to each template.

Now, inside your other template files, instead of calling the `main` function, call `main`, `cart`, or `blog`, as needed. For instance, if you have a template named *post.scala.html* for your blog posts, that template file can call the `blog` function to use *blog.scala.html* as a wrapper, as shown here:

```
@(title: String, blogPostContent: String)

@* call the blog.scala.html 'wrapper' template *@
@blog(title) {

  @blogPostContent

}
```

A product page in an ecommerce store might invoke the *cart.scala.html* wrapper template, as shown here:

```
@(title: String, product: Product)

@cart(title) {

  <!-- add code here to display the Product ... -->

}
```

Because Scala template files are compiled to functions, wrapping a template with boilerplate code for a particular section of a website is very simple.

A quick example

If you followed the steps in Recipe 2, you can test this approach by following these steps:

1. Create a *blog.scala.html* template file as described in this recipe. Modify its `<title>` tag, or add an `<h1>` tag so you can differentiate its output from the *main.scala.html* file.
2. Edit the *people.scala.html* template created in Recipe 2, and change `@main` to `@blog` in that file.
3. Assuming you still have the Play server running, reload the *http://localhost:8080/people* URL. You should see the wrapper output from your *blog.scala.html* wrapper in the `<h1>` or `<title>` tags you added.

4. Creating Reusable Code Blocks in Templates

Problem

You have repetitive code in a template and want to create a function in the template to keep from having to repeat the code, i.e., to keep it DRY ("Don't Repeat Yourself").

Solution

Play lets you create reusable code blocks in a template. These code blocks work like functions to help keep your code DRY.

As an example, the following template file named *links.scala.html* has a reusable code block named `displayLiLink`. It takes two parameters, a URL and a description, and outputs those parameters inside an anchor tag inside an `<li>` tag:

```
@()

@displayLiLink(url: String, description: String) = {
  <li><a href="@url">@description</a></li>
}
```

```
@main("Websites") {

  <h1>Websites</h1>

  <ul>
    @displayLiLink("http://google.com", "Google")
    @displayLiLink("http://yahoo.com", "Yahoo")
    @displayLiLink("http://alvinalexander.com", "My Website")
  </ul>

}
```

The `displayLiLink` function is called three times within the `<ul>` section shown. Ignoring extra whitespace, this results in the following code being output to the browser:

```
<ul>
<li><a href="http://google.com">Google</a></li>
<li><a href="http://yahoo.com">Yahoo</a></li>
<li><a href="http://alvinalexander.com">My Website</a></li>
</ul>
```

If you've been following along with the previous recipes, you can demonstrate this by making a few additions to your project. First, create a new file named *links.scala.html* in the *views* directory with the contents shown.

Then add this new route to your *conf/routes* file:

```
GET    /links      controllers.Application.links
```

Then add this method to the *controllers/Application.scala* file:

```
def links = Action {
  Ok(views.html.links())
}
```

Now, when you access the *http://localhost:8080/links* URL in your browser, you should see the list of links from the *links.scala.html* template.

Discussion

Reusable code blocks like this are easy to create and use in Play templates. The hardest part about creating and using them can be knowing when to use the special `@` symbol.

As the Play templates documentation indicates, the `@` character marks the beginning of a Scala statement. For simple expressions, Play is able to determine the end of your code block, so there is no need for a closing symbol. This was shown in the lines where the `displayLiLink` block was called:

```
@displayLiLink("http://google.com", "Google")
```

The reusable code block showed that you may need to use the `@` character in multiple places. In the example, the `@` character is used to define the code block, and then used to identify the variables inside the code block:

```
@displayLiLink(url: String, description: String) = {
  <li><a href="@url">@description</a></li>
}
```

As the Play templates documentation states, "Because the template engine automatically detects the end of your code block by analyzing your code, this syntax only supports simple statements. If you want to insert a multi-token statement, explicitly mark it using brackets." The documentation demonstrates this in the following example:

```
Hello @(customer.firstName + customer.lastName)!
```

I've found this approach useful in many situations, such as when you want to return a simple text string from a reusable code block, as shown in the `@title` code block in the following example:

```
@(items: List[String])

@title = @{ "Your Shopping Cart" }

@cart(title) {

  <h1>@title</h1>

  <ul>
    @items.map { item =>
      <li>@item</li>
    }
  </ul>

}
```

Though that's a trivial example, it demonstrates how to properly return a string literal from a reusable code block. Attempting to define the code block as follows results in an error:

```
@* intentional error *@

@title = "Your Shopping Cart"
```

On a related note, if you need to display an `@` character in your HTML output, just enter it twice. This is necessary when you need to print an email address:

```
<p>al@@example.com</p>
```

You can also call functions in regular Scala classes from templates. This is shown in the next recipe.

5. Calling Scala Functions from Templates

Problem

You want to call a function in a Scala class from a template.

Solution

You can easily call Scala functions from Play templates. For instance, given a class named `HtmlUtils` in the *controllers* package:

```
package controllers

object HtmlUtils {

  def li(string: String) = <li>{string}</li>
  def anchor(url: String, description: String) =
      <a href={url}>{description}</a>

}
```

you call the `anchor` method from a Play template like this:

```
<p>Here's a link to @HtmlUtils.anchor("http://google.com", "Google")</p>
```

Discussion

Notice that no import statement was required in the template because the `HtmlUtils` class was defined in the *controllers* package. If the `HtmlUtils` class was defined in a different package, like this:

```
package com.alvinalexander.htmlutils

object HtmlUtils {

  def li(string: String) = <li>{string}</li>
  def anchor(url: String, description: String) =
      <a href={url}>{description}</a>

}
```

you would need an import statement in the template, like this:

```
@* just after the first line of your template *@
@import com.alvinalexander.htmlutils.HtmlUtils

@* somewhere later in the code ... *@
<p>Here's a link to @HtmlUtils.anchor("http://google.com", "Google")</p>
```

Because `HtmlUtils` is an object, you can change the `import` statement to import its methods into scope, and then just call the `anchor` method (without prefixing it with the `HtmlUtils` object name), as shown here:

```
@* import HtmlUtils._ *@
@import com.alvinalexander.htmlutils.HtmlUtils._

@* just call 'anchor' *@
<p>Here's a link to @anchor("http://google.com", "Google")</p>
```

Passing functions into templates

Although this recipe demonstrates how to call functions on an object, it's worth mentioning that you can also pass functions into your templates as template parameters.

For instance, in the `Application` controller you can define the following methods:

```
def sayHello = <p>Hello, via a function</p>

def functionDemo = Action {
  Ok(views.html.function(sayHello))
}
```

The function named `functionDemo` calls a Play template named *function.scala.html*, and passes the `sayHello` method to it as a variable. Because `sayHello` returns output of type `scala.xml.Elem`, the *function.scala.html* template should be defined like this:

```
@(callback: => scala.xml.Elem)

@main("Hello") {

  @callback

}
```

If you're not familiar with Scala's functional programming (FP) support, the parameter that's passed into the template is defined like this:

```
callback: => scala.xml.Elem
```

This means that this is a function (or method) that takes no arguments, and returns a `scala.xml.Elem`.

If you created the example shown in Recipe 1, you can demonstrate this by adding the following route to the *conf/routes* file:

```
GET   /function    controllers.Application.functionDemo
```

After creating the *app/views/function.scala.html* template, adding the code to the *app/controllers/Application.scala* and the *conf/routes* files, when you access the *http://localhost:8080/function* URL in your browser, you'll see the "Hello, via a function" output.

See Also

Recipe 1, "Creating a "Hello, World" Project"

6. Creating a Widget and Including It in Pages

Problem

You want to create one or more "widgets" (components) and include those in your web pages. This might include a shopping cart widget in an online store, a list of recent blog posts in a blog, or any other reusable content you want to display.

Solution

This solution is similar to the previous recipe on calling methods in a Scala object from a template. You can use that approach to emit HTML code from a function, or you can place your widget code in another template file. The latter approach is shown in this recipe.

To demonstrate this approach, imagine that you're creating a "product detail" page for a shopping cart. As a result, you'll have a template file named *product.scala.html*. For this simple example, the template will include two main components, (a) the information you want to output about the current product and (b) a shopping cart widget that will be shown at the side of the page:

```
@(product: (String, String), items: List[String])

@* product.scala.html *@

@main(product._1) {

  <!-- include the shopping cart widget -->
  @cartWidget(items)

  <!-- a description of the current product -->
  <div style="padding:10px; margin:10px;">
    <h1>@product._1</h1>
    <div id="product_info">
      <p>@product._2</p>
    </div>
  </div>

}
```

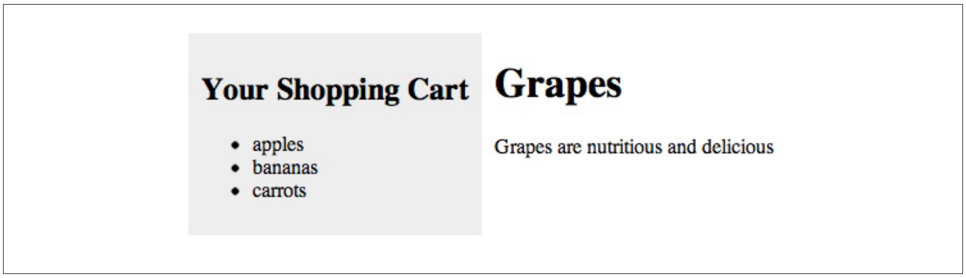
In this case the `@cartWidget(items)` code refers to another template file named *cartWidget.scala.html*. Its code looks like this:

```
@(items: List[String])

<div style="background-color:#eee; padding:10px; margin:10px; float:left">
  <h2>Your Shopping Cart</h2>
  <ul>
    @items.map { item =>
      <li>@item</li>
    }
  </ul>
</div>
```

This template takes a `List[String]` that represents the items in the current shopping cart, and `items` was passed to `@cartWidget` in the *product.scala.html* file.

Assuming that you add this code to your project as described in the Discussion, the combination of these templates will result in the output shown in Figure 4.


Your Shopping Cart

- apples
- bananas
- carrots

Grapes

Grapes are nutritious and delicious

Figure 4. The cart widget is included with the product content

Discussion

An important concept to remember about Play is that template files are compiled down to Scala functions. As a result, calling them—and therefore including their output in another template—is a simple process.

If you followed along with the steps in Recipe 1, you can add this code to that same project to demonstrate and experiment with it. First, create the *product.scala.html* and *cartWidget.scala.html* template files in the *app/views* directory.

Next, add this method to the *Application.scala* file in the *app/controllers* directory:

```
def product = Action {
  val grapes = ("Grapes", "Grapes are nutritious and delicious")
  val cart = List("apples", "bananas", "carrots")
  Ok(views.html.product(grapes, cart))
}
```

Then add this route to the *conf/routes* file:

```
GET    /product     controllers.Application.product
```

With these files in place, go back to your browser and access the *http://localhost:8080/product* URL, and you should see the results shown in Figure 4.

See Also

You can clone the source code for this recipe from GitHub.

7. Using CoffeeScript and LESS

Problem

You want to use popular web technologies like CoffeeScript and LESS CSS in your Play application.

Solution

CoffeeScript is a popular replacement for JavaScript, and LESS is a popular replacement for writing CSS. It's easy to use both technologies in your Play applications, as shown in the following sections.

Using CoffeeScript

To use CoffeeScript in a Play application, follow these steps:

1. If your application doesn't already have an *app/assets* folder, create it.
2. Inside the *assets* folder, create a *scripts* folder for your CoffeeScript files.
3. Place your custom CoffeeScript files inside the new *scripts* folder.
4. Assuming you created a file named *main.coffee* in the *scripts* folder, Play will automatically compile your CoffeeScript file to JavaScript, and you can then include the JavaScript file in your templates (such as *main.scala.html*) like this:

    ```
    <script src="@routes.Assets.at("scripts/main.js")" /></script>
    ```

Notice that the file *main.js* is generated from your *main.coffee* file.

That's all you have to do. You can test this by following those steps, then placing this code in the *main.coffee* file:

```
alert "Hello, world"
```

If you add the `<script>` line shown to the `<head>` section of your *main.scala.html* template file, just access one of your URLs in your browser that uses this template. When you reload the page, you should see a JavaScript alert dialog displayed.

Using LESS

Using LESS is also easy. Just follow these steps to begin using it:

1. If your application doesn't already have an *app/assets* folder, create it.
2. Inside the *assets* folder, create a folder named *stylesheets*.
3. Inside that folder, create your custom LESS files. For instance, create a file named *myapp.less*.
4. Play will compile your LESS source code to regular CSS. Assuming you named your file *myapp.less*, a corresponding file named *myapp.css* will be generated, and you can including it in your Play templates like this:

   ```
   <link rel="stylesheet" media="screen"
         href="@routes.Assets.at("stylesheets/myapp.css")">
   ```

To test this, put the following code into a file named *myapp.less* in the *app/assets/stylesheets* folder:

```
@color: red;

h1 {
  color: @color;
}
```

Then add this `<link>` tag into the `<head>` section of your main template wrapper file, i.e., *app/views/main.scala.html*:

```
<link rel="stylesheet" media="screen"
      href="@routes.Assets.at("stylesheets/myapp.css")">
```

When you add an `<H1>` tag to a template that includes this CSS file, your `<H1>` tags will be displayed in a red color.

See Also

- The CoffeeScript website
- The LESS CSS website

8. Creating a Simple Form

Problem

You want to get started creating forms in a Play Framework application.

Solution

Creating a new Play form is roughly a seven-step process:

1. Add new routes to *app/conf/routes*.
2. Create a template for your form.
3. Add a form mapping to your controller.
4. Add a form to your controller.
5. Create a controller action to display the form.
6. Create a second controller action to handle the form submission.
7. Create any model code necessary to work with the form, including classes to model the domain (`Person`, `Address`, `Stock`, etc.), and data access objects.

I'll demonstrate these steps by creating a form to add a new `Stock` in a sample Play application. A `Stock` consists of a stock market symbol and company name, such as `Stock("GOOG", "Google, Inc.")`. When completed, the form will look like Figure 5.

You can follow the steps in this recipe, or clone my Play "Form Validations" Project.

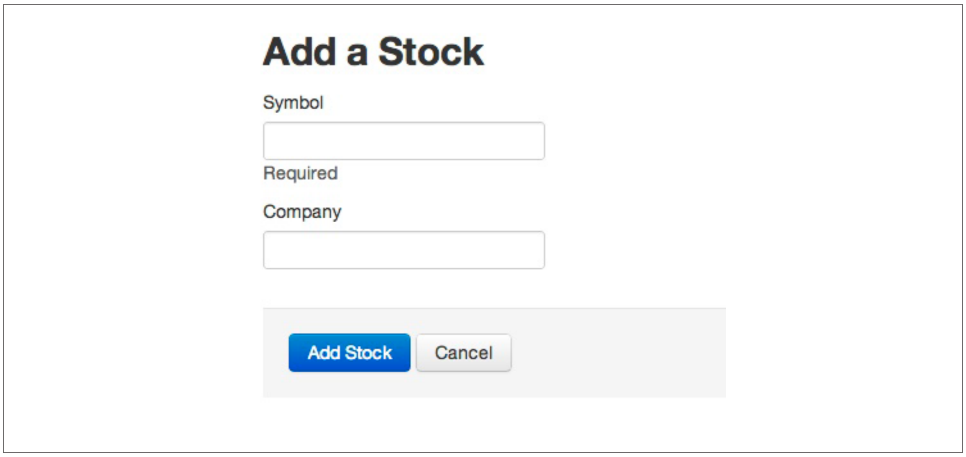


Figure 5. The form to add a new stock

To get started, first create a new Play application with the `play new` command:

```
$ play new Stocks
```

Answer Play's questions, and then move into the directory it creates for you.

Add a route to app/conf/routes

Next, edit the *app/conf/routes* file and add two entries to the end of the file. The `add` entry will be used to display the new form at the URL *http://localhost:8080/stocks/add*. When this form is submitted, it will submit its contents using the POST method to the `save` action:

```
# stocks
GET    /stocks/add      controllers.Stocks.add
POST   /stocks/save     controllers.Stocks.save
```

Create a template for your form

Next, create a Play template for the form. Save the following code to a file named *form.scala.html* in a new directory named *app/views/stock*:

```
@(stockForm: Form[Stock])

@import helper._
@import helper.twitterBootstrap._

@main("Add Stock") {

  <h2>Add a Stock</h2>

  @helper.form(action = routes.Stocks.save, 'class->"form-inline") {

    @inputText(
      stockForm("symbol"),
      '_label -> "Symbol",
      'class -> "control-label"
    )

    @inputText(
      stockForm("company"),
      '_label -> "Company",
      'class -> "control-label"
    )

    <div class="form-actions">
      <input type="submit" class="btn btn-primary" value="Add Stock">
      <a href="@routes.Application.index" class="btn">Cancel</a>
    </div>

  }

}
```

This is a basic Play form template, with a bit of CSS added to make the form look a little better. Values like `'_label` and `'class` are described in Table 4 in Recipe 10, but as you might guess, they represent the label and CSS class for each field.

Add a form mapping in your controller

Now it's time to start creating a `Stocks` controller. Create a file named *Stocks.scala* in the *app/controllers* directory with the following stub code:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models.Stock

object Stocks extends Controller {

}
```

(If you want to skip ahead, the complete code for this class is shown in the Discussion.)

Now, when the form in the *form.scala.html* template is submitted, the form data will be sent to the `save` method in the `Stocks` class. When this happens, the two fields in the form will be represented by a `Map`. For instance, if the user types in the information for Google's stock, the `Map` will look like this:

```
Map("symbol" -> "GOOG",
    "company" -> "Google")
```

The approach to handling this form data in Play is to create a *form mapping* as a field in the `Stocks` controller class. The following mapping declares that the `symbol` field can't be empty—it's a required field—but the `company` field is optional:

```
object Stocks extends Controller {

  // the new form mapping field
  val formMapping = mapping(
    "symbol" -> nonEmptyText,
    "company" -> optional(text)
  )
  (Stock.apply)(Stock.unapply)

}
```

The type of the `formMapping` field is `play.api.data.Mapping[models.Stock]`.

The `Stock.apply` method is used to construct a new `Stock` instance from the mapping, such as when a new `Stock` instance is created. The `Stock.unapply` method is used in the opposite case, when you want to create a mapping from an existing `Stock` object, such as when editing an existing object.

Add a form in your controller

Next, create a `Form` instance from the mapping. Add the following line of code just below the `formMapping`:

```
val stockForm: Form[Stock] = Form(formMapping)
```

The code for the `Mapping` and `Form` are often included in one statement, but I've separated them here to demonstrate the steps and types.

Create a controller action to display the form

Next, create an action in the controller to display the form. This action was referred to as `controller.Stocks.add` in the *conf/routes* files, so name it `add`:

```
def add = Action {
  Ok(views.html.stock.form(stockForm))
}
```

This is a normal Play method that implements an `Action`. It simply displays the template named *app/views/stock/form.scala.html*, passing the `stockForm` to the template.

Create a second controller action to handle the form submission

Next, you need a controller action to handle the form submission. The following code shows the pattern to handle a form submission:

```
def save = Action { implicit request =>
  stockForm.bindFromRequest.fold(

    // (1) on a validation error go back to the form
    errors => BadRequest(views.html.stock.form(errors)),

    // (2) on success create the stock, go to another page
    stock => {
      Stock.save(stock)
      Redirect(routes.Stocks.add)
    }
  )
}
```

The `save` method receives the HTTP request from the form, and the `bindFromRequest` method binds the `stockForm` to the data received in the request. This process is called *binding* the request to the form.

Because the logic of evaluating a form results in two possible branches—failure or success—the `fold` method is a good choice to handle this. In the failure case (#1), when the form validation process results in an error, call the `BadRequest` function, giving it a reference to the form so it can redisplayed.

In the success case (#2), a new `Stock` object is created, so save it to the database, and then forward the user to whatever page you want to display next. To keep this example small, the code redirects users to the same Add a Stock page, but you can forward them to any template you define.

Create any model code necessary to work with the form

For this form, create a case class named `Stock` and a corresponding companion object. To do this, first create a *models* folder under the *app* folder, and then create a *Stock.scala* file in the *models* folder.

Rather than creating a full DAO at this time, just create a simple `Stock` object with a `save` method that provides a little debugging output. Put this code in the *Stock.scala* file:

```
package models

case class Stock(symbol: String, company: Option[String])

object Stock {

  def save(stock: Stock) {
    println(s"Would have created stock: $stock")
  }

}
```

In your real-world code, you would implement this `save` method as shown in Recipe 11, "Selecting from a Database with Anorm", but to keep this example relatively simple, I avoided that extra code.

One extra step

I followed one extra step in my example to create a decent-looking form. As described in the Discussion, I added some "Twitter Bootstrap" code to my form to make it look a little better. If you follow this additional step, your "Add a Stock" form should look like Figure 6.

Testing

To test all of the new code, start the Play console from the root directory of your project:

```
$ play
```

and then start the Play server:

```
[Stocks] $ run 8080
```

You should now be able to access the form at the *http://localhost:8080/stocks/add* URL.

When your form is running, you should be able to successfully submit it as long as you supply text for the `symbol` field. The `company` field is optional, but if you don't supply text for the `symbol` field when you submit the form, you should see the "This field is required" error message shown in Figure 6.

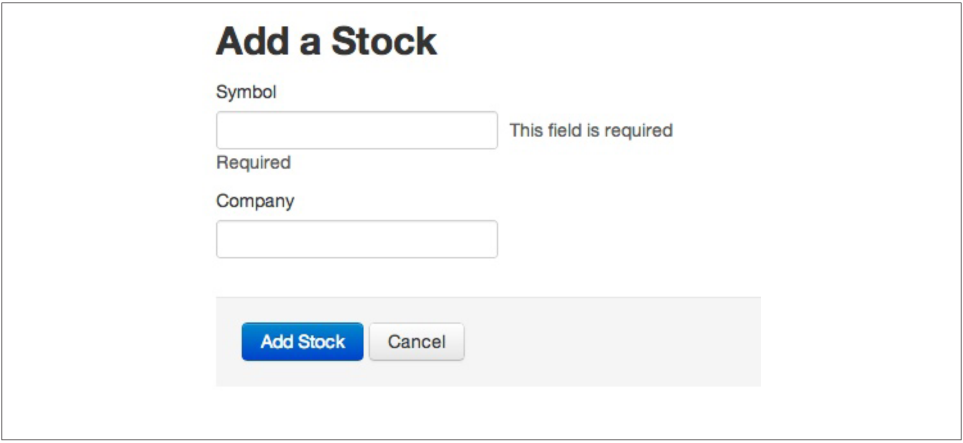


Figure 6. When the form is submitted without a Symbol value, an error message is displayed

Discussion

The complete code for the `Stocks` controller class is shown here for your convenience:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models.Stock

object Stocks extends Controller {

  val formMapping = mapping(
      "symbol" -> nonEmptyText,
      "company" -> optional(text)
    )(Stock.apply)(Stock.unapply)

  val stockForm: Form[Stock] = Form(formMapping)

  def add = Action {
    Ok(views.html.stock.form(stockForm))
  }

  /**
* Handle the 'add' form submission.
*/
  def save = Action { implicit request =>
    stockForm.bindFromRequest.fold(
      // (1) on a validation error go back to the form
      errors => BadRequest(views.html.stock.form(errors)),
          // (2) on success create the stock, go to another page
          stock => {
            Stock.save(stock)
            Redirect(routes.Stocks.add)
          }
        )
      }

    }
```

As mentioned in the Solution, the `Form` and `Mapping` are often combined in one step, like this:

```
    val stockForm: Form[Stock] = Form(
      mapping(
        "symbol" -> nonEmptyText,
        "company" -> optional(text)
      )(Stock.apply)(Stock.unapply)
    )
```

Defining the form mapping is typically the most difficult part of creating a new form. As you'll see in Recipe 9, "Validating a Form", form field validations are added to this code as well, so in real-world code, the mapping can get more complex.

When you define a `Mapping`, Play provides a number of data manipulation helpers that you can use to define form fields. These helpers are defined in the play.api.data.Forms object. Table 1 in the next recipe shows many of the helpers that are available in Play 2.1.1.

Generating Play forms fast

Years ago I realized that most initial form development is driven by your database design. For instance, most of the code shown in these Anorm recipes can be generated from `stocks` database table. Realizing this, I created a "CRUD Generator" tool named Cato to generate the initial "CRUD" (Create-Read-Update-Delete) source code for my applications. Because Cato is language-independent and template-driven, I was able to create Cato templates for the Play Framework that let me rapidly create Play forms. See this video demonstration of how I can create a complete initial set of Play CRUD forms for a real-world database table in just over seven minutes.

Using Twitter Bootstrap

Twitter Bootstrap is a frontend framework to help make cross-platform web development easier. If you ever started a new web development project and wished there was a standard set of CSS definitions for web forms (and a few other tools), Bootstrap may be what you're looking for.

At the time of this writing, Play's support for Bootstrap is in flux. The latest release of Bootstrap is version 2.3.2, but Play 2.1.1 supports Bootstrap 1.4.x, so using that version is demonstrated here.

Probably the easiest way to use the Twitter Bootstrap 1.4.x release is to copy the files that are needed from the "forms" sample project that ships with Play. You'll find the *forms* project folder under the *samples* directory of your Play installation folder. There are both Scala and Java versions of this project, so use the Scala version.

Within the *forms* project, switch to the *public/stylesheets* folder. From that folder, copy the *bootstrap.css* and *main.css* files, then paste them into the same directory in your Play project. If you already have files with these names, be careful about overwriting them.

Once you've copied those files into your project, add these lines of code to the `<head>` section of your template wrapper file, e.g., the default *main.scala.html*. The line to include the *main.css* file may already exist:

```
<link rel="stylesheet" media="screen"
      href="@routes.Assets.at("stylesheets/bootstrap.css")">
<link rel="stylesheet" media="screen"
      href="@routes.Assets.at("stylesheets/main.css")">
```

As shown in the *form.scala.html* template in this recipe, you'll also need to include this line of code in your form template files:

```
@import helper.twitterBootstrap._
```

Now, when you develop your forms, they should be styled with the Twitter Bootstrap CSS. Some of this styling is shown in Figure 7.

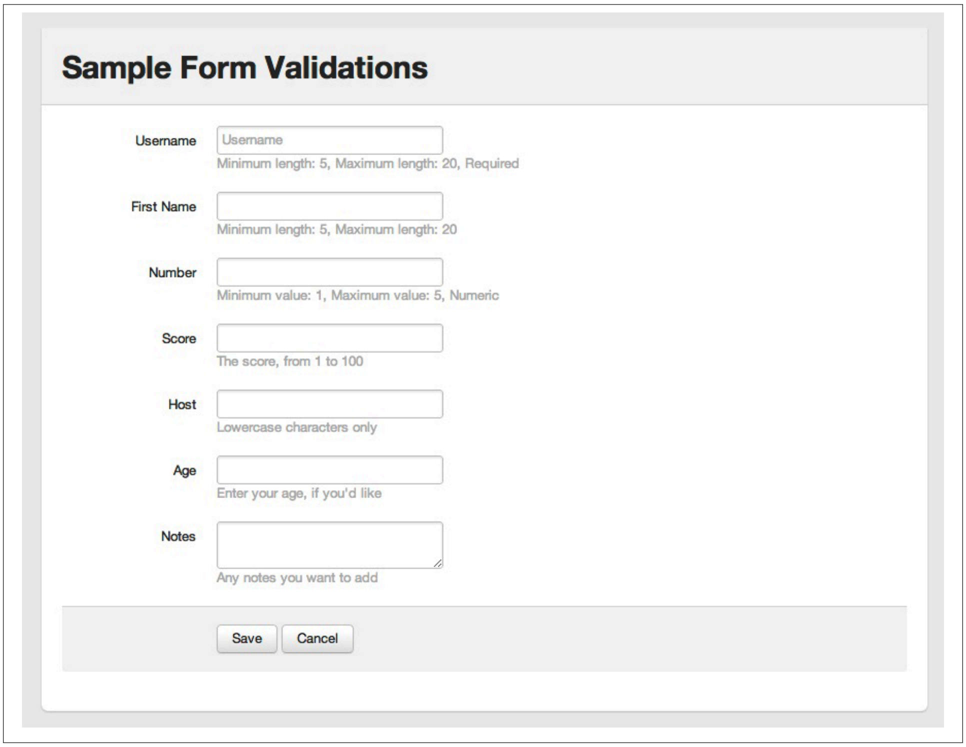
9. Validating a Form

Problem

You want to validate the fields in a form in a controller method to make sure the data matches your constraints before attempting to save the form data to a database.

Solution

When you define a `Mapping`, Play provides a number of data manipulation helpers that you can use to define form fields. These helpers come from the play.api.data.Forms object. Table 1 shows many of the helpers that are available in Play 2.1.1.


Sample Form Validations

Username
Username
Minimum length: 5, Maximum length: 20, Required
First Name
Minimum length: 5, Maximum length: 20
Number
Minimum value: 1, Maximum value: 5, Numeric
Score
The score, from 1 to 100
Host
Lowercase characters only
Age
Enter your age, if you'd like
Notes
Any notes you want to add
Save
Cancel

Figure 7. A sample form styled with Twitter Bootstrap (and a little additional CSS)

Table 1. Common Play data manipulation helpers

Data manipulation helper	Description
`boolean`	A mapping for a Boolean field, such as a checkbox.
`date`	A mapping for a date field.
`email`	A mapping for an email field.
`ignored`	A field in your form that should be ignored for validation purposes.
`list`	A repeated mapping, such as when you prompt a user with an email field and a “verify email address” field.
`longNumber`	A mapping for a numeric field. Uses a `Long` type.
`nonEmptyText`	A mapping for a required text field.
`number`	A mapping for a numeric field (`Int`).
`optional`	Makes the mapping optional.
`single`	A mapping for a single value.
`sqlDate`	A mapping for a date field, mapped as a `sql.Date`.
`text`	A mapping for a text field.

See the play.api.data.Forms object documentation for additional mappings.

The following list of example form fields shows different ways that these helpers can be used:

```
"readEula"  -> boolean,
"date"      -> date("yyyy-MM-dd"),
"email"     -> email,
"id"        -> ignored(1234),
"stocks"    -> list(text),
"addresses" -> list(email),
"username"  -> nonEmptyText,
"username"  -> nonEmptyText(5),  // requires a minimum of five characters
"count"     -> number,
"company"   -> optional(text),
"number"    -> optional(number),
"notes"     -> text,
"password"  -> text(minLength = 10),
```

More examples of these constraints are demonstrated in this recipe.

As described in Table 2, the Play Framework also defines constraints in the `play.api.data.validation.Constraints` object.

Table 2. Constraints from the play.api.data.validation.Constraints object

Constraints' method	Description
`min(minValue: Int): Constraint[Int]`	A constraint to specify a minimum value for an `Int`.
`max(maxValue: Int): Constraint[Int]`	Specify a maximum value for an `Int`.
`minLength(length: Int): Constraint[String]`	Specify a minimum length constraint for a `String`.
`maxLength(length: Int): Constraint[String]`	Specify a maximum length constraint for `String`.
`nonEmpty: Constraint[String]`	Create a "required" constraint for a `String`.
`pattern(` `regex: Regex,` `name: String,` `error: String): Constraint[String]`	Create a regular expression constraint for a `String`.

Although you can use the `min`, `max`, `minLength`, and `maxLength` methods, the Play classes offer some conveniences, so you can just put the `min` and `max` values in parentheses of the data manipulation helpers, as shown in these examples:

```
"username" -> nonEmptyText(5, 20),   // 5 to 20 characters
"password" -> nonEmptyText(8),       // at least eight characters
```

The following example `Form` demonstrates most of the built-in validations, including how to specify a pattern while validating a text field:

```
val mongoForm = Form(
  mapping(
    "username" -> nonEmptyText(5, 20),
        "firstName" -> text(5, 20),
        "middleInitial" -> optional(text),
        "email" -> email,
        "number" -> number(1, 5),
        "host" ->
          text.verifying(pattern("[a-z]*".r, "Lowercase chars only", "Error")),
        "age" -> optional(number),
        "longNumber" -> longNumber,
        "optionalNumber" -> optional(number),
        "date" -> date("yyyy-MM-dd"),    // java.util.Date
        "password" -> nonEmptyText(8),
        "readEula" -> checked("Please accept the terms of the EULA"),
        "yesNoSelect" -> text,           // treat select/option as 'text'
        "yesNoRadio" -> text,            // treat radio buttons as 'text'
        "stocks" -> list(text),
        "notes" -> optional(text),
        "ignored" -> ignored("foo")      // static value
      )(Mongo.apply)(Mongo.unapply)
    )
  )
```

When the built-in validators aren't enough, you can define your own constraints using the `verifying` method, both on individual fields (as shown on the `host` field) and at the form level.

For instance, in my Finance application, I check to see whether a stock is already in the database before I attempt to add it. I can make that check either at the field level or at the form level. The following code demonstrates how to use `verifying` at the *field level* to test whether the stock is already in the database:

```
val stockForm: Form[Stock] = Form(
  mapping(
    "symbol" -> nonEmptyText.verifying(
                  "D'oh - Stock already exists!",
                  Stock.findBySymbol(_) == 0),
    "company" -> optional(text))
    (Stock.apply)(Stock.unapply)
)
```

In this case the validation is at the field level, so this field will be validated at the same time as all other fields in the form. The downside of this approach is that the `Stock.findBySymbol` method will be called every time the form is submitted, and the upside is that if the stock is already in the database, I can tell the user about this at the same time as I tell him about any other field errors. (This is trivial in this example, but can be important in a larger form or on a busy website.)

The following code demonstrates how to perform the same verification test at the *form level*:

```
val stockForm: Form[Stock] = Form(
  mapping(
    "symbol" -> nonEmptyText,
    "company" -> optional(text))
    (Stock.apply)(Stock.unapply)
    verifying("D'oh - Stock already exists!", fields => fields match {
      // this block creates a 'form' error.
      // this only gets called if all field validations are okay.
      case Stock(i, s, c) =>  Stock.findBySymbol(s) == 0
  })
)
```

As the comments mention, a `verifying` method included here will only be called when all of the field-level validations pass. Therefore, this hit on the database will only happen when the form has otherwise been filled out properly.

As you probably suspected, the `Stock.findBySymbol` method that is invoked in these `verifying` calls returns the count of the number of records found in the `stocks` database table that has the same symbol. Using Anorm, that method looks like this:

```
def findBySymbol(symbol: String): Long = {
  if (symbol.trim.equals("")) return 0
  DB.withConnection { implicit c =>
    val firstRow =
      SQL("SELECT COUNT(*) AS c FROM stocks WHERE symbol = {symbol}")
      .on('symbol -> symbol.toUpperCase)
      .apply
      .head
    firstRow[Long]("c")  // returns the count
  }
}
```

Discussion

The best way to demonstrate these validations is with an example form. To that end, I've created a PlayFormValidations project that you can clone from GitHub. This project creates the form shown in Figure 8. It demonstrates common validations, and how you can control the form appearance with the template file and form mappings.

The form in Figure 8 was created by putting the following code in *conf/routes*:

```
# home page
GET     /                       controllers.Application.index

# validation examples
GET     /validations/add        controllers.ValidationsController.add
POST    /validations/save       controllers.ValidationsController.save

# map static resources from the /public folder to the /assets URL path
GET     /assets/*file           controllers.Assets.at(path="/public", file)
```

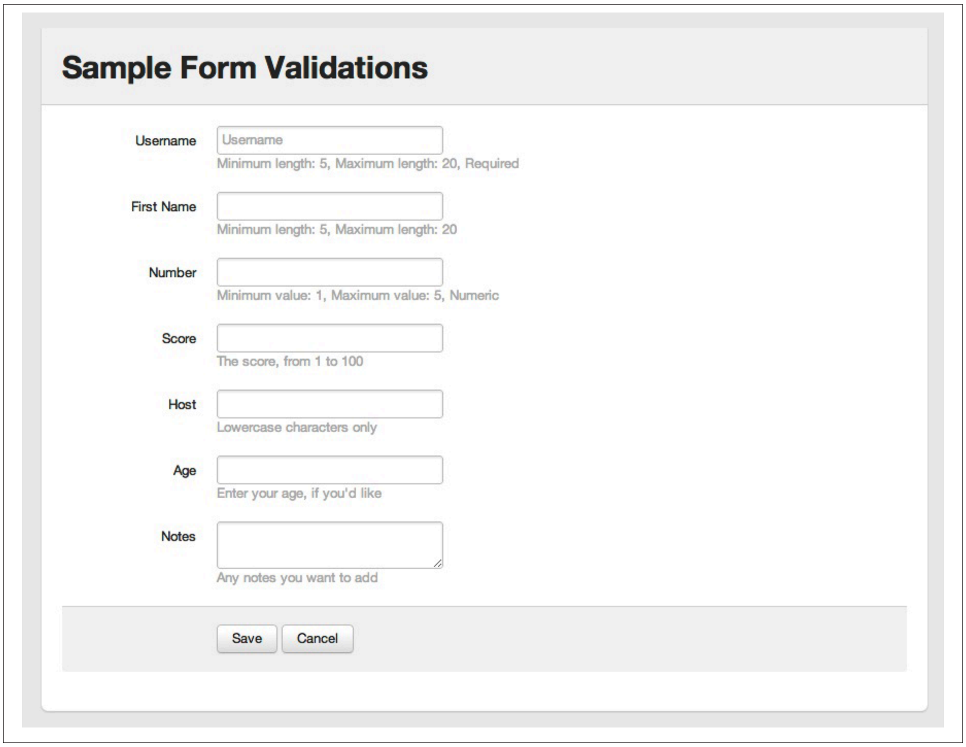

Sample Form Validations
Username
Username
Minimum length: 5, Maximum length: 20, Required
First Name
Minimum length: 5, Maximum length: 20
Number
Minimum value: 1, Maximum value: 5, Numeric
Score
The score, from 1 to 100
Host
Lowercase characters only
Age
Enter your age, if you'd like
Notes
Any notes you want to add
Save
Cancel

Figure 8. An example form that demonstrates common form field validations

The template file for the form is *app/views/validationsform.scala.html*:

```
@(validationsForm: Form[Validations])

@import helper._
@import helper.twitterBootstrap._

@main("Sample Form Validations") {

  @* this block of code will display form-level errors *@
  @if(validationsForm.hasErrors) {
    <div class="alert-message">
      <p>There were one or more errors with the form:</p>
      <ul>
        @validationsForm.errors.map { error =>
          <li>@error.message</li>
        }
      </ul>
    </div>
  }
  @helper.form(action = routes.ValidationsController.save) {

    @* demonstrates a textfield, label, and placeholder text *@
    @inputText(validationsForm("username"), '_label -> "Username",
               'placeholder -> "Username")

    @* you can use placeholders on these fields as well *@
    @inputText(validationsForm("firstName"), '_label -> "First Name")
    @inputText(validationsForm("number"), '_label -> "Number")
    @inputText(validationsForm("score"), '_label -> "Score",
               '_help -> "The score, from 1 to 100")
    @inputText(validationsForm("host"), '_label -> "Host")
    @inputText(validationsForm("age"), '_label -> "Age",
               '_help -> "Enter your age, if you'd like")
    @textarea(validationsForm("notes"), '_label -> "Notes",
               '_help -> "Any notes you want to add")

    <div class="form-actions actions">
      <input type="submit" class="btn btn-primary" value="Save">
      <a href="@routes.Application.index" class="btn">Cancel</a>
    </div>

  }

}
```

The template demonstrates several different useful techniques, including setting placeholder text on the Username field, and supplying help text for several other fields. Refer to Figure 8 to see the help text that Play automatically generates for the fields I haven't manually supplied, including the First Name, Number, and Host fields.

The form validation code is in *app/controllers/ValidationsController.scala*:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models.Validations
import play.api.data.validation.Constraints._
import scala.util.matching.Regex

object ValidationsController extends Controller {

  val x = pattern("".r, "", "")

  val validationsForm = Form(
    mapping(
      "username" -> nonEmptyText(5, 20),
      "firstName" -> text(1, 20),
      "number" -> number(1, 5),
      "score" -> number.verifying(min(1), max(100)),
      "host" -> nonEmptyText.verifying(pattern("[a-z]+".r,
                  "One or more lowercase characters", "Error")),
      "age" -> optional(number),
      "notes" -> optional(text)
    )(Validations.apply)(Validations.unapply)
      verifying("If age is given, it must be greater than zero", model =>
        model.age match {
          case Some(age) => age < 0
          case None => true
        }
      )
  )

  def add = Action {
    Ok(views.html.validationsform(validationsForm))
  }

  /**
* Handle the 'add' form submission.
*/
  def save = Action { implicit request =>
    validationsForm.bindFromRequest.fold(
      errors => BadRequest(views.html.validationsform(errors)),
      stock => {
        // would normally do a 'save' here
        Redirect(routes.ValidationsController.add)
      }
    )
  }

}
```

Finally, the corresponding model is in *app/models/Validations.scala*:

```
package models

case class Validations (
  username: String,
  firstName: String,
  number: Int,
  score: Int,
  host: String,
  age: Option[Int],
  notes: Option[String]
)
```

Once you have all the files in place, start the Play server as usual. I run it on port 8080:

```
$ play

[PlayFormValidations] $ run 8080
```

Then access the form at the *http://localhost:8080/validations/add* URL.

Field-level validations will result in error messages right next to the field where the error occurred, and because of the way the template is defined, form-level errors will be displayed above the form. For instance, the following `verifying` code on the form mapping is a form-level validation:

```
verifying("If age is given, it must be greater than zero", model =>
  model.age match {
    case Some(age) => age < 0
    case None => true
  }
)
```

As the text implies, it checks to see if an `age` is given, and if the `age` is given, it must be greater than zero. When this validation error is triggered, the error message that's displayed above the form looks like Figure 9.

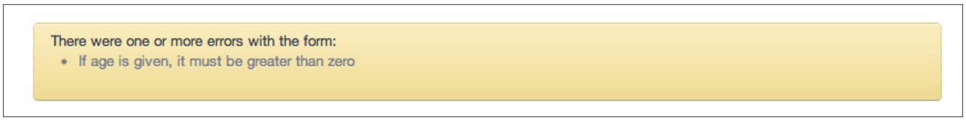

There were one or more errors with the form:
- If age is given, it must be greater than zero

Figure 9. A form-level validation error message

This error message is displayed due to the following block of code, which is included in the template, above the form:

```
@* this block of code will display form-level errors *@
@if(validationsForm.hasErrors) {
  <div class="alert-message">
    <p>There were one or more errors with the form:</p>
```

```
        <ul>
          @validationsForm.errors.map { error =>
            <li>@error.message</li>
          }
        </ul>
      </div>
    }
```

This recipe demonstrates a number of different methods to validate a form. To experiment with this code on your own system, clone my GitHub project.

10. Displaying and Validating Common Play Form Elements

Problem

You want to use common HTML elements in a Play Framework form, such as a text field, textarea, drop-down list, checkbox, buttons, etc., and it would be helpful to see examples of how they are created and used.

Solution

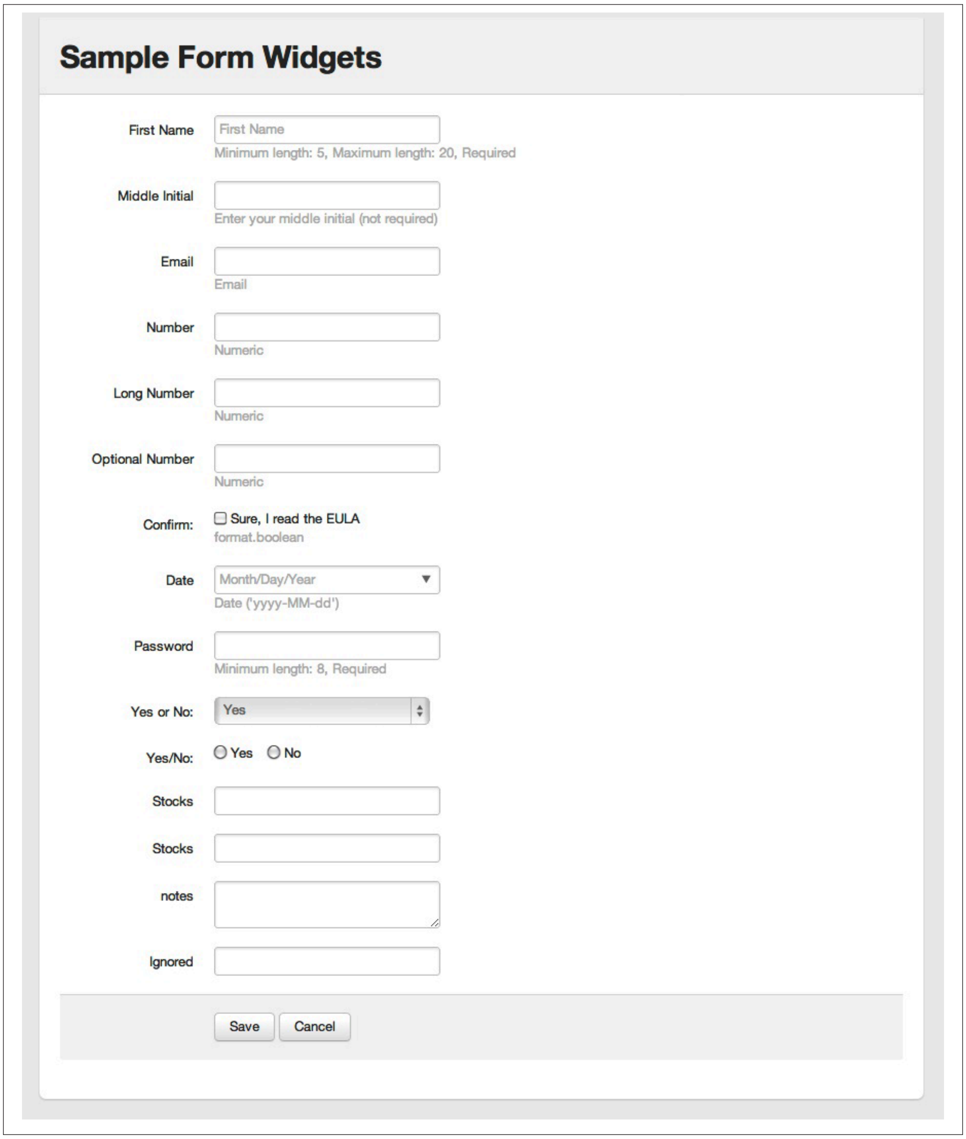
The easiest way to demonstrate the common Play form widgets is to create a form that has at least one of each widget type. The "mongo" form shown in Figure 10 shows all the built-in widgets types.

As discussed in previous recipes, you create this form by adding the following components to your project:

- A form template
- A form controller class
- A model class

The easiest way to use this code is to clone my "Mongo Form" project.

I created the form template with the filename *app/views/mongoform.scala.html*. Its contents are:


Sample Form Widgets

First Name	First Name Minimum length: 5, Maximum length: 20, Required
Middle Initial	Enter your middle initial (not required)
Email	Email
Number	Numeric
Long Number	Numeric
Optional Number	Numeric
Confirm:	☐ Sure, I read the EULA format.boolean
Date	Month/Day/Year Date ('yyyy-MM-dd')
Password	Minimum length: 8, Required
Yes or No:	Yes
Yes/No:	○ Yes ○ No
Stocks	
Stocks	
notes	
Ignored	

Figure 10. This large form demonstrates common form widgets

```
@(mongoForm: Form[Mongo])

@import helper._
@import helper.twitterBootstrap._

@main("Sample Form Widgets") {
  @helper.form(action = routes.MongoController.save) {

    @* demonstrates a textfield, label, and placeholder text *@
    @inputText(mongoForm("username"), '_label -> "First Name",
              'placeholder -> "First Name")

    @inputText(mongoForm("middleInitial"),
               '_label -> "Middle Initial",
               '_help -> "Enter your middle initial (not required)")

    @* email and number fields *@
    @inputText(mongoForm("email"), '_label -> "Email")
    @inputText(mongoForm("number"), '_label -> "Number")
    @inputText(mongoForm("longNumber"), '_label -> "Long Number")
    @inputText(mongoForm("optionalNumber"), '_label -> "Optional Number")

    @* checkbox *@
    @checkbox(mongoForm("readEula"), '_label -> "Confirm:",
              '_text -> "Sure, I read the EULA")

    @* date *@
    @inputDate(mongoForm("date"), '_label -> "Date")

    @* password *@
    @inputPassword(mongoForm("password"), '_label -> "Password")

    @* select/option field *@
    @select(mongoForm("yesNoSelect"), options("yes"->"Yes", "no"->"No"),
            '_label -> "Yes or No:")

    @* radio buttons *@
    @inputRadioGroup(mongoForm("yesNoRadio"), options("yes"->"Yes", "no"->"No"),
                     '_label -> "Yes/No:")

    @* request user enter multiple words *@
    @helper.repeat(mongoForm("stocks"), min = 2) { stockField =>
      @inputText(stockField, '_label -> "Stocks")
    }

    @textarea(mongoForm("notes"))

    @* 'ignored' field (static content) *@
    @inputText(mongoForm("ignored"), '_label -> "Ignored")

    <div class="form-actions actions">
      <input type="submit" class="btn btn-primary" value="Save">
      <a href="@routes.Application.index" class="btn">Cancel</a>
    </div>

  }

}
```

This template refers to a *main.scala.html* wrapper template file:

```
@(title: String)(content: Html)

<!DOCTYPE html>

<html>
  <head>
    <title>@title</title>
    <link rel="stylesheet" media="screen"
          href="@routes.Assets.at("stylesheets/bootstrap.css")">
    <link rel="stylesheet" media="screen"
          href="@routes.Assets.at("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.at("images/favicon.png")">
    <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")"
            type="text/javascript"></script>
  </head>

  <body>
  <div class="container">

    <div class="content">

      <div class="page-header">
        <h1>@title</h1>
      </div>

      <div class="row">
        <div class="span14">
          @content
        </div>
      </div>

    </div>

    <footer>
      <p>
      </p>
    </footer>

  </div>
  </body>

</html>
```

To validate and process the form, I created a file named *app/controllers/MongoController.scala*:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models.Mongo

object MongoController extends Controller {

  val mongoForm = Form(
    mapping(
      "username" -> nonEmptyText(5, 20),
      "middleInitial" -> optional(text),
      "email" -> email,
      "number" -> number,
      "longNumber" -> longNumber,
      "optionalNumber" -> optional(number),
      "date" -> date("yyyy-MM-dd"),    // java.util.Date
      "password" -> nonEmptyText(8),
      "readEula" -> checked("Please accept the terms of the EULA"),
      "yesNoSelect" -> text,           // treat select/option as 'text'
      "yesNoRadio" -> text,            // treat radio buttons as 'text'
      "stocks" -> list(text),
      "notes" -> optional(text),
      "ignored" -> ignored("foo")      // static value
    )(Mongo.apply)(Mongo.unapply)
  )

  def add = Action {
    Ok(views.html.mongoform(mongoForm))
  }

  /**
* Handle the 'add' form submission.
*/
  def save = Action { implicit request =>
    mongoForm.bindFromRequest.fold(
      errors => BadRequest(views.html.mongoform(errors)),
      stock => {
        // would normally do a 'save' here
        Redirect(routes.MongoController.add)
      }
    )
  }

}
```

The Mongo form class is at *app/models/Mongo.scala*, and is defined like this:

```
package models

import java.util.Date

case class Mongo (
  username: String,
  middleInitial: Option[String],
  email: String,
  number: Int,
  longNumber: Long,
  optionalNumber: Option[Int],
  date: Date,
  password: String,
  readEula: Boolean,
  yesNoSelect: String,
  yesNoRadio: String,
  stocks: List[String],
  notes: Option[String],
  ignored: String
)
```

Once you have all the files in place, start the Play server as usual. I run it on port 8080:

```
$ play

[MongoForm] $ run 8080
```

You can now access the form at the *http://localhost:8080/mongo/add* URL.

Discussion

The code in this recipe demonstrates three essential things related to Play forms:

- How to create each widget in a template file using Play's predefined helpers.
- How to map and validate each widget.
- How to create a model to match the mapping.

An important part of this recipe is understanding how to configure the proper mapping for each widget. I included some extra rows in the template to demonstrate many of the common form mappings, including `text`, `nonEmptyText`, `optional(text)`, and more difficult mappings like checkboxes, the select/option control, and radio buttons. For those more difficult controls, the examples show the following:

- An `@checkbox` widget maps to a `checked` validation.
- The `@select` widget maps to a `text` validation.
- The `@inputRadioGroup` maps to a `text` validation.

The input helpers are defined in the package object of Play's *views.html.helper* package. Table 3 provides a brief description of the common helper objects.

Table 3. Common Play helper objects

Play helper object	Description
`checkbox`	An HTML input checkbox.
`form`	Creates an HTML form.
`input`	A generic HTML input.
`inputDate`	An HTML5 date input.
`inputFile`	An HTML file input.
`inputPassword`	An HTML password input field.
`inputRadioGroup`	An HTML radio group.
`inputText`	An HTML text input field.
`select`	An HTML select/option field.
`textarea`	An HTML textarea.

As shown in the examples, you can set "input helper" options on the fields, using an object known as a `FieldConstructor`. Options you can set are shown in Table 4.

Table 4. Play input helper options

Field constructor option	Description
`_error -> "Error, error!"`	Use a custom error message for the field.
`_help -> "(mm-dd-yyyy)"`	Show custom help text.
`_id -> "stock-form"`	Create a CSS ID for the top `<DL>` element.
`_label -> "Symbol:"`	Use a custom label for the field. (This is very common.)
`_showConstraints -> true`	Set to `true` to show the field constraints, or `false` to hide them.
`_showErrors -> true`	Set to `false` to hide errors on the field.

As mentioned, this example uses some custom CSS that's based on the Twitter Bootstrap project. The templates use two CSS files that I copied from the Play *samples/form* project, and then modified. See Recipe 8 for a discussion about using Twitter Bootstrap 1.4 with Play 2.1.1.

See Also

- The easiest way to use all of this code is to clone my GitHub project
- Play's predefined helpers

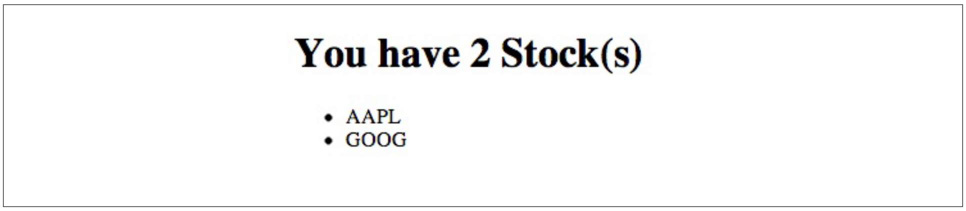
11. Selecting from a Database with Anorm

Problem

You want to select data from a database using the Play's built-in Anorm library.

Solution

There are several different ways to write SQL `SELECT` methods using Anorm, and each approach will be shown here. When you've finished this recipe, you'll have all of the code needed to display a list of stocks at a URL, as shown in Figure 11.


You have 2 Stock(s)

- AAPL
- GOOG

Figure 11. The result of selecting all the stocks from the database

To make it easy to learn Anorm, I created a project you can clone from GitHub. It includes the code from all of the Anorm recipes in this chapter.

One-time configuration

The first thing you'll need for this recipe is a MySQL database table named `stocks` with this definition:

```
create table stocks (
  id int auto_increment not null,
  symbol varchar(10) not null,
  company varchar(32),
  primary key (id),
  constraint unique index idx_stock_unique (symbol)
);
```

You'll also need some sample data, so insert a few records into the table:

```
INSERT INTO stocks (symbol, company) VALUES ('AAPL', 'Apple');
INSERT INTO stocks (symbol, company) VALUES ('GOOG', null);
```

Next, create a new Play application, as shown in Recipe 1. (Use the `play new` command.)

Now you need to connect your Play application to the MySQL database. To do this, edit the *conf/application.conf* file, and add these lines to the "Database configuration" section of that file:

```
db.default.url="jdbc:mysql://localhost:8889/stocks"
db.default.driver=com.mysql.jdbc.Driver
db.default.user=root
db.default.pass=root
```

My database is named `stocks`, and I use MAMP, which runs MySQL on port `8889` by default. Change these settings as needed for your server.

You also need to add MySQL as a dependency to your project. To do this, edit the *project/Build.scala* file in your project, and add MySQL as a dependency to the `appDependencies` variable:

```
val appDependencies = Seq(
  // Add your project dependencies here,
  jdbc,
  anorm,
  "mysql" % "mysql-connector-java" % "5.1.25"
)
```

Now that your Play application is ready to connect to your MySQL database, it's time to write the code to display the results of a SQL `SELECT` statement.

Steps to displaying the results of a SQL SELECT statement

The steps required to display the results of a SQL `SELECT` query at a new URL are:

1. Create a template to show the results.
2. Add an entry to the *conf/routes* file to bind the template to a controller method.
3. Create a `Stocks` controller.
4. Create a `Stock` model class and a corresponding `Stock` object (a companion object).

Create a template to show the results

To create the view shown in Figure 11, first create a *stock* folder under the *app/views* folder. Then create a *list.scala.html* file under the *stock* folder with these contents:

```
@(stocks: List[Stock])

@main("Stocks") {

  <h1>You have @stocks.size Stock(s)</h1>

  <div>
  <ul>
    @stocks.map { stock =>
      <li>
        @stock.symbol
      </li>
    }
    </ul>
    </div>

}
```

This template receives a `List[Stock]` and calls the `main` wrapper template to display the `Stock` symbols in a bulleted list.

Configure the route

To list the stocks at the */stocks* URI, create this entry in the *conf/routes* file:

```
GET   /stocks     controllers.Stocks.list
```

Create a Stocks controller class

Now create a `Stocks` controller with a `list` method to match the route:

```
package controllers

import play.api._
import play.api.mvc._
import views._
import models._

object Stocks extends Controller {

  def list = Action {
    Ok(html.stock.list(Stock.selectAll()))
  }

}
```

The `list` method gets a `List` of `Stock` objects from the `selectAll` method of a `Stock` object, and passes that list to the *list.scala.html* template file in the *app/views/stock* folder.

Create a Stock model class and companion object

For the `SELECT` query (and all other SQL queries), you'll need a `Stock` model class, which you can define as a simple case class.

The Anorm standard is to create database methods in the companion object of the model class, so create a `Stock` object in the same file. To select records from the database, you need a "select all" method, which I named `selectAll`.

To implement this code, create the *app/models* folder, then create a file in the *models* folder named *Stock.scala*, with this source code:

```
package models

case class Stock (val id: Long,
                  var symbol: String,
                  var company: Option[String])
```

```
object Stock {

  import play.api.db._
  import play.api.Play.current

  // create a SqlQuery for all of the "select all" methods
  import anorm.SQL
  val sqlQuery = SQL("select * from stocks order by symbol asc")

  def selectAll(): List[Stock] = DB.withConnection { implicit connection =>
    sqlQuery().map ( row =>
      Stock(row[Long]("id"),
            row[String]("symbol"),
            row[Option[String]]("company"))
    ).toList
  }

}
```

If you've written JDBC code before, this code is somewhat similar to using a `ResultSet`. The `selectAll` method executes the `sqlQuery` (which is an instance of `anorm.SqlQuery`), calls the `map` method on the `sqlQuery`, creates a new `Stock` object from each `Row` in the results, and returns the result as a `List[Stock]`.

Notice that the `company` field is declared as an `Option[String]` in the case class, and is used similarly in the `selectAll` method. This is how you handle optional fields, which may be `null` in the database.

Access the URI

When you access the */stocks* URI in your browser, such as *http://localhost:8080/stocks*, you should see the result shown in Figure 11, a list of stocks in the `stocks` database table.

Discussion

There are several other ways to write `SELECT` queries with Anorm. A second approach uses Scala's pattern-matching capability to create `Stock` instances based on each row:

```
import anorm.Row

def selectAll() : List[Stock] = {
  DB.withConnection { implicit connection =>
      sqlQuery().collect {
          case Row(id: Int, symbol: String, Some(company: String)) =>
                  Stock(id, symbol, Some(company))
          case Row(id: Int, symbol: String, None) =>
                  Stock(id, symbol, None)
      }.toList
   }
}
```

Two case statements are needed because the company field may be null. If a company name is found, the first case statement is matched, but if it's null, the second statement is matched.

A third approach uses the Anorm Parser API, which gives you a DSL that you can use to define a RowParser to build a Stock object from each row:

```
import anorm._
import anorm.SqlParser._

// uses the Parser API
// stock is an instance of anorm.RowParser[models.Stock]
val stock = {
  get[Long]("id") ~
  get[String]("symbol") ~
  get[Option[String]]("company") map {
    case id~symbol~company => Stock(id, symbol, company)
  }
}

import play.api.db._
import play.api.Play.current

def selectAll(): List[Stock] = DB.withConnection { implicit c =>
  sqlQuery.as(stock *)
}
```

All three of these approaches return the same result, a List[Stock], so they can be used interchangeably.

Here's the complete source code for an *app/models/Stock.scala* file that shows all three approaches, including all the necessary import statements:

```
package models

case class Stock (val id: Long,
                  var symbol: String,
                  var company: Option[String])

object Stock {

  import play.api.db._
  import play.api.Play.current

  // create a SqlQuery for all of the "select all" methods
  import anorm.SQL
  import anorm.SqlQuery
  val sqlQuery = SQL("select * from stocks order by symbol asc")

  /**
* SELECT * (VERSION 1)
* -------------------------------------------------------------------------
```

```
*/
  import play.api.Play.current
  import play.api.db.DB
  def selectAll1(): List[Stock] = DB.withConnection { implicit connection =>
    sqlQuery().map ( row =>
      Stock(row[Long]("id"),
            row[String]("symbol"),
            row[Option[String]]("company"))
    ).toList
  }

  /**
* SELECT * (VERSION 2)
* ------------------------------------------------------------------------
*/
  import anorm.Row
  def selectAll2() : List[Stock] = {
    DB.withConnection { implicit connection =>
        sqlQuery().collect {
            case Row(id: Int, symbol: String, Some(company: String)) =>
                    Stock(id, symbol, Some(company))
            case Row(id: Int, symbol: String, None) =>
                    Stock(id, symbol, None)
            case foo => println(foo)
                      Stock(1, "FOO", Some("BAR"))
        }.toList
     }
  }

  /**
* SELECT * (VERSION 3)
* ------------------------------------------------------------------------
*/
  import anorm._
  import anorm.SqlParser._

  // a parser that will transform a JDBC ResultSet row to a Stock value
  // uses the Parser API
  // http://www.playframework.org/documentation/2.0/ScalaAnorm
  val stock = {
    get[Long]("id") ~
    get[String]("symbol") ~
    get[Option[String]]("company") map {
      case id~symbol~company => Stock(id, symbol, company)
    }
  }

  import play.api.db._
  import play.api.Play.current
  // method requires 'val stock' to be defined
  def selectAll3(): List[Stock] = DB.withConnection { implicit c =>
    sqlQuery.as(stock *)
    }

  }
```

You can experiment with this code by cloning my Play Stocks project from GitHub.

See Also

- The Play Framework "Accessing an SQL Database" page
- The Play Anorm page
- My Play Stocks project

12. Inserting Data into a Database with Anorm

Problem

You want to save data to a database using the built-in Play Framework "Anorm" library.

Solution

Follow the "One-time configuration" steps from Recipe 11 to create a MySQL `stocks` database and connect your Play project to it. You'll also need the *app/controllers/Stocks.scala* and *app/models/Stock.scala* files from that project. Then follow these steps:

1. Create a data entry form (template) to let a user add a new stock.
2. Add the necessary entries to the *conf/routes* file.
3. Create a `Form` in the `Stocks` controller to match the template.
4. Create methods in the `Stocks` controller to (a) display the form, and (b) validate and accept it when it's submitted.
5. Create an `insert` method in the `Stock` object in *app/models/Stock.scala*.

Create a data entry form

The data entry form for a `Stock` is simple, and is shown in Figure 12.

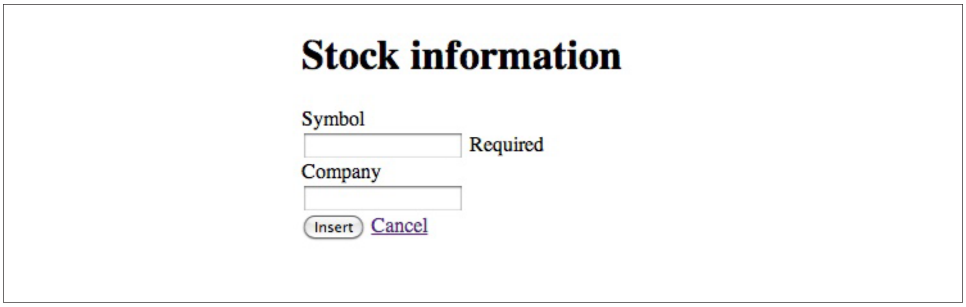


Figure 12. The "Add Stock" form created in this recipe

To create the template, create the *app/views/stock* folder if it doesn't already exist. Then create a *form.scala.html* template file in that folder with these contents:

```
@(stockForm: Form[Stock])

@import helper._

@main("Stocks") {

  @helper.form(action = routes.Stocks.submit) {

     <h1>Stock information</h1>

     @inputText(
       stockForm("symbol"),
       '_label -> "Symbol"
     )

     @inputText(
       stockForm("company"),
       '_label -> "Company"
     )

    <div class="actions">
      <input type="submit" class="btn primary" value="Insert">
      <a href="@routes.Stocks.list" class="btn">Cancel</a>
    </div>

  }

}
```

This template (which compiles to a function) takes a `Form[Stock]` as a parameter. The template calls the `main` wrapper template, as usual. The `@helper.form` and `@inputText` fields are described in Recipes 8 through 10, but if you've used a templating system before, they probably look familiar. `@helper.form` creates an HTML `<form>` element, and the `@inputText` fields render HTML `<input type="text">` fields.

When the form is submitted, the form `action` shows that it will be submitted to the `submit` method in the `Stocks` controller class.

Add two entries to the routes file

Next, when creating an "add" form like this, you need to add two entries to the *conf/routes* file. Assuming you created the "list" action in Recipe 11, add the two new lines at the end of this file:

```
GET   /stocks        controllers.Stocks.list

# new
GET   /stocks/add    controllers.Stocks.add
POST  /stocks        controllers.Stocks.submit
```

With this configuration, the "add" form will appear at the */stocks/add* URI, and will be displayed by the `add` method of the `Stocks` controller. When the form is submitted, it will be submitted with the `POST` method to the `submit` method of the `Stocks` controller.

Create the Form in the controller

Next, you need a Play `Form` that maps to the fields in the *form.scala.html* template:

```
// defines a mapping that will handle Stock values
val stockForm: Form[Stock] = Form(
  mapping(
    "symbol" -> nonEmptyText,
    "company" -> optional(text))
    ((symbol, company) => Stock(0, symbol, company))
    ((s: Stock) => Some((s.symbol, s.company)))
)
```

As mentioned in Recipe 11, the `symbol` field is required, so it's defined as `nonEmptyText` here. (Data for this field will be a `String` like `AAPL`.)

The two lines of code at the end of the form define `apply` and `unapply` methods that are used to create a new `Stock` object from the form data, or convert an existing `Stock` into use by a form, respectively:

```
((symbol, company) => Stock(0, symbol, company))
((s: Stock) => Some((s.symbol, s.company)))
```

Create the necessary controller class actions

With the `Form` in place, two actions are needed in the controller: an `add` method to display the template, and a `submit` method to handle the form submission.

Here's the complete code for the `Stocks` controller (*app/controllers/Stocks.scala*), which includes these methods and the `stockForm`:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import play.api.data.validation.Constraints._
import views._
import models._

object Stocks extends Controller {

  // defines a mapping that will handle Stock values
  val stockForm: Form[Stock] = Form(
    mapping(
      "symbol" -> nonEmptyText,
      "company" -> optional(text))
      ((symbol, company) => Stock(0, symbol, company))
      ((s: Stock) => Some((s.symbol, s.company)))
  )

  def list = Action {
    Ok(html.stock.list(Stock.selectAll3()))
  }

  /**
* Display the 'add' form.
*/
  def add = Action {
    Ok(html.stock.form(stockForm))
  }

  /**
* Handle form submission.
*/
  def submit = Action { implicit request =>
    stockForm.bindFromRequest.fold(
      errors => BadRequest(html.stock.form(errors)),  // back to form
      stock => {
        // todo: this code assumes that Stock.save always succeeds
        val result = Stock.save(stock)
        println(s"INSERT succeeded, id = $result")
        Redirect(routes.Stocks.list)
      }
    )
  }

}
```

Displaying the form with the `add` method is simple: just pass the `stockForm` to the *form.scala.html* template in the *app/views/stock* folder while calling the `Ok` method to display the template.

The `submit` method is also a Play `Action`. It takes an implicit `request` variable, then attempts to bind the data the user submitted to the `stockForm`. If this initial binding process succeeds—the user input passes the form validations—the flow of control passes to the `stock` match in the `fold` method, where the `Stock.save` method is called. Assuming that succeeds, the browser is redirected to the *list.scala.html* template created in Recipe 11 by calling the `list` method of the `Stocks` controller. If you didn't copy the code from that recipe, redirect the user back to the *form.scala.html* template instead by calling the controller's `add` method.

If the binding process fails, the `errors` case in the `fold` method is invoked, and *form.scala.html* is redisplayed using Play's `BadRequest` method. Any errors—such as not providing a stock symbol—are displayed on the data entry form.

Notice that neither the `stockForm` nor the `submit` method attempt to determine whether the given stock is already in the database. More robust validation code is included in my GitHub project, which checks to see if a stock exists in the database before attempting to insert it.

Create a Stock companion object

The final piece of the puzzle that's needed is an Anorm `save` method in the `Stock` companion object in the *app/models/Stock.scala* file:

```
object Stock {

  def save(stock: Stock): Option[Long] = {
    val id: Option[Long] = DB.withConnection { implicit c =>
      SQL("insert into stocks (symbol, company) values ({symbol}, {company})")
      .on('symbol -> stock.symbol.toUpperCase,
          'company -> stock.company
      ).executeInsert()
    }
    id
  }

}
```

Note that this is a normal SQL `INSERT` query, with some Anorm code wrapped around it. If you've used a library like Spring JDBC, this may seem familiar.

The syntax in the `on` method refers to field names as `'symbol` and `'company` is just one way to write this query. You can enclose the field names in double quotes, if you prefer:

```
.on("symbol" -> stock.symbol.toUpperCase,
    "company" -> stock.company
```

Preceding a variable name with a single quote creates an instance of a `Symbol`. See the Scala Symbol Scaladoc for more information.

If, as in this example, you're inserting data into a table that has an autogenerated `Long` primary key (an `auto_increment` field in MySQL), `executeInsert` returns the value of the `id` field. You can also use `executeUpdate` here. It returns an `Int` indicating the number of fields affected, which is hopefully always `1` for an `INSERT`. This is good for SQL `UPDATE` queries, but I prefer to use `executeInsert`, if possible.

Note that this code does not include a `try/catch` block. As a result, it can throw a MySQL integrity constraint violation if you attempt to insert a stock symbol that already exists. You can see this in your browser by attempting to insert the same stock symbol more than once.

Test the form

With all of this code in place, go to your browser and access the */stocks/add* URI, e.g. *http://localhost:8080/stocks/add*. Once your code is compiled, you should see the form shown in Figure 12. When you enter valid data, the form submission process should succeed, and redirect you to the */stocks* URI, which was implemented in Recipe 11. If you skipped that recipe, just redirect the form back to itself.

If you leave the `Symbol` field blank and submit the form, the form submission process will fail, and the form will be redisplayed, showing the error that the `Symbol` field is a required field.

See Also

My Play Stocks project

13. Deleting Records in a Database Table with Anorm

Problem

You want to delete records in a database table using Anorm.

Solution

Assuming you followed the "One-time configuration" steps from Recipe 11 to create a MySQL `stocks` database and connect your Play project to it, you can use the following `delete` method in a `Stock` object in *app/models/Stock.scala* to delete a record, given the primary key (`id`) of the stock to be deleted:

```
object Stock {
  def delete(id: Long): Int = {
    DB.withConnection { implicit c =>
      val numRowsDeleted = SQL("DELETE FROM stocks WHERE id = {id}")
        .on('id -> id)
        .executeUpdate()
      numRowsDeleted
    }
  }
}
```

In this example, a maximum of one record should be deleted, so `numRowsDeleted` should be `1` (it succeeded) or `0` (it failed).

Ignoring error handling, this method can be called from a `Stocks` controller (*app/controllers/Stocks.scala*) method like this:

```
def delete(id: Long) = Action {
  Stock.delete(id)
  Redirect(routes.Stocks.list)
}
```

In that example, the code ignores the `Int` that is returned, but it can also be handled:

```
def delete(id: Long) = Action {
  val numRowsDeleted = Stock.delete(id)
  // add logic based on numRowsDeleted ...
}
```

See Also

My Play Stocks project includes all of the code needed to implement a complete "delete" solution, including the route, template, controller, and model code needed.

14. Updating Records in a Database Table with Anorm

Problem

You want to update records in a database table using Anorm.

Solution

Assuming you followed the "One-time configuration" steps from Recipe 11 to create a MySQL `stocks` database and connect your Play project to it, write an `update` method in your `Stock` object (the companion object in the *app/models/Stock.scala* file). You can use the following `update` method to update records, given the primary key (`id`) field and a new `Stock` object to replace the old one:

```
object Stock {
  def update(id: Long, stock: Stock): Boolean = {
    DB.withConnection { implicit c =>
      SQL("update stocks set symbol={symbol}, company={company} where id={id})")
      .on('symbol -> stock.symbol,
          'company -> stock.company,
          'id -> id
      ).executeUpdate() == 1
    }
  }
}
```

The syntax that refers to the field names as `'symbol`, `'company`, and `'id` in the `on` method call is just one way to write this query. You can enclose the field names in double quotes, if you prefer:

```
.on("symbol" -> stock.symbol,
    "company" -> stock.company,
    "id" -> id
```

Preceding a variable name with a single quote creates an instance of a `Symbol`. See the Scala Symbol Scaladoc for more information.

The `executeUpdate` method returns the number of rows affected by the query, so in this case, it should return a value of `1`. If the result is `1`, the method returns `true`, otherwise it returns `false`.

See Also

My Play Stocks project includes all of the code needed to implement a complete "update" solution, including the route, template, controller, and model code.

15. Testing Queries Outside of Play

Problem

You want a simple, convenient way to test your Anorm SQL queries.

Solution

At least two developers have created approaches to let you test Anorm queries outside of a full-blown Play application:

- Timothy Klim's anorm-without-play project
- HendraWijaya's anorm-examples project

Both projects are normal SBT projects, so they're easy to use. I cloned Timothy Klim's project, added the MySQL dependency to the `libraryDependencies` field in the *build.sbt* file:

```
"mysql" % "mysql-connector-java" % "5.1.25"
```

deleted the *Main.scala* file that comes with the project:

```
$ rm src/main/scala/Main.scala
```

and then created a file named *StockQueriesTests.scala* in the root directory of the SBT project with these contents:

```
import java.sql.Connection
import scalikejdbc.ConnectionPool
import java.util.Date
import anorm._
import anorm.SqlParser._

object StockQueryTests extends App {

  Class.forName("com.mysql.jdbc.Driver")
  ConnectionPool.singleton("jdbc:mysql://localhost:8889/stocks",
                           "root", "root")

  object DB {
    def withConnection[A](block: Connection => A): A = {
      val connection: Connection = ConnectionPool.borrow()
      try {
        block(connection)
      } finally {
        connection.close()
      }
    }
  }

  case class Stock (
    val id: Long,
    var symbol: String,
    var company: Option[String]
  )

  // the DAO
  object Stock {

    // SELECT
    def selectAll() : List[Stock] = {
      DB.withConnection { implicit connection =>
          SQL("select * from stocks")().collect {
              case Row(id: Int, symbol: String, Some(company: String)) =>
                      Stock(id, symbol, Some(company))
              case Row(id: Int, symbol: String, None) =>
                      Stock(id, symbol, None)
              case foo => println("selectAll Error: Found something else: "
                                  + foo)
                        Stock(1, "FOO", Some("BAR"))
          }.toList
       }
    }

    // INSERT
    def save(stock: Stock) {
      DB.withConnection { implicit c =>
        SQL("insert into stocks (symbol, company) values ({symbol}, {company})")
        .on('symbol -> stock.symbol,
            'company -> stock.company
        ).executeUpdate()
      }
    }

    // DELETE
    def delete(symbol: String): Int = {
      DB.withConnection { implicit c =>
        val nRowsDeleted = SQL("DELETE FROM stocks WHERE symbol = {symbol}")
          .on('symbol -> symbol)
          .executeUpdate()
        nRowsDeleted
      }
    }

  } // Stock

  // INSERT
  println("ADD NETFLIX:")
  Stock.save(Stock(0, "NFLX", Some("Netflix")))
  println(Stock.selectAll())

  // DELETE
  println("DELETE NETFLIX:")
  println(Stock.delete("NFLX"))
  println(Stock.selectAll())

}
```

Running this object with the `sbt run` command verifies that all of the queries work as expected.

To make this more convenient, you can also run the `sbt eclipse` command to generate the files needed for Eclipse, and then run your code through Eclipse.

Discussion

You can add SQL debugging to your project by adding the following configuration lines to your project's *conf/application.conf* file:

```
db.default.logStatements=true
logger.com.jolbox=DEBUG
```

Those lines tell Play to print the actual SQL statements that are executed when a URL is accessed to the Play console.

16. Deploying a Play Framework Project

Problem

You want to deploy your Play Framework project to a production environment.

Solution

There are several ways to deploy your Play application to a production server:

- Use the Play `dist` command to create a ZIP file with everything needed to run your application.
- Get your project's source code onto your production server, and "stage" it.

Both approaches are shown here.

Use the Play dist command

You can build a complete binary version of your application with the Play `dist` command. To do this, start the Play command-line tool in the root directory of your project, and then run the `dist` command:

```
[Finance] $ dist

(output omitted ...)
Your application is ready in dist/dist/finance-1.0-SNAPSHOT.zip

[success]
```

This creates a ZIP file that contains everything you need, including a `start` command, *README* file, and all the JAR files needed to run the application.

To run your application on a production server, copy the ZIP file to the server, unzip it, make the `start` command executable, and then run it. For example, once you have a ZIP file, such as *finance-1.0-SNAPSHOT.zip* on a production server, the process looks like this:

```
$ unzip finance-1.0-SNAPSHOT.zip

Archive:  finance-1.0-SNAPSHOT.zip
   creating: finance-1.0-SNAPSHOT/
   creating: finance-1.0-SNAPSHOT/lib/
  inflating: finance-1.0-SNAPSHOT/lib/org.scala-lang.scala-library-2.10.0.jar
  inflating: finance-1.0-SNAPSHOT/lib/play.play_2.10-2.1.1.jar

  many lines of output skipped here ...
  inflating: finance-1.0-SNAPSHOT/lib/finance_2.10-1.0-SNAPSHOT.jar
  inflating: finance-1.0-SNAPSHOT/start
  inflating: finance-1.0-SNAPSHOT/README

$ cd finance-1.0-SNAPSHOT

$ ls -al
total 16
drwxr-xr-x   5 Al  staff   170 May 16 12:28 .
drwxr-xr-x   4 Al  staff   136 May 16 12:30 ..
-rw-r--r--   1 Al  staff   151 Apr  2 20:25 README
drwxr-xr-x  56 Al  staff  1904 May 16 12:28 lib
-rw-r--r--   1 Al  staff  3000 May 16 12:28 start

$ chmod +x start

$ ./start
Play server process ID is 14124
[info] play - database [default] connected at jdbc:mysql://localhost:8889/stocks
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0.0.0.0:9000
```

The `start` script is a simple shell script that executes a `java` command:

```
#!/usr/bin/env sh

exec java $* -cp "`dirname $0`/lib/*" play.core.server.NettyServer↵
 `dirname $0`
```

As you can see from the script, you don't even need Scala installed on your production server, just Java. This makes it easy to deploy your application to all sorts of application server environments, including your own servers as well as servers from Heroku, Amazon, Google, and many more.

Stage the application

A second way to deploy your application to a production environment is to copy your Play application's source code to a production server, where you can run the application by "staging" it. This lets you start the application from the operating system command line, which also lets you automate the starting of the application.

As a simple example, imagine that you've used Git or another tool to get your application's source code onto your production server. Once you've done that, run the following `play` command from your operating system command line to stage your application:

```
$ play clean compile stage

[info] Loading global plugins from /Users/Al/.sbt/plugins
[info] Loading project definition from project
[info] Updating
[info] Done updating.
```

```
[info] Compiling 9 Scala sources and 1 Java source to target/scala-2.10/↵
classes...
[success] Total time: 19 s
[info] Packaging target/scala-2.10/finance_2.10-1.0-SNAPSHOT-sources.jar ...
[info] Done packaging.
[info] Wrote scala-2.10/finance_2.10-1.0-SNAPSHOT.pom
[info] Generating Scala API documentation for main sources to
       target/scala-2.10/api...
[info] Packaging target/scala-2.10/finance_2.10-1.0-SNAPSHOT.jar ...
[info] Done packaging.
[info] Scala API documentation generation successful.
[info] Packaging target/scala-2.10/finance_2.10-1.0-SNAPSHOT-javadoc.jar ...
[info] Done packaging.
[info]
[info] Your application is ready to be run in place: target/start
[info]
[success] Total time: 6 s, completed May 16, 2013 12:36:52 PM
```

As one of the last output lines indicates, you can now run your application from the command line as `target/start`:

```
$ target/start

Play server process ID is 14365
[info] play - database [default] connected at jdbc:mysql://localhost:8889/stocks
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0.0.0.0:9000
```

I prefer using the `dist` approach, but staging the application can also be useful.

Discussion

If you want to run your application in *production mode* in your development or test environments, you can run the application by using the `start` command from the Play console prompt (instead of the `run` command):

```
[MyApp] $ start

(Starting server. Type Ctrl+D to exit logs, the server will remain in background)

Play server process ID is 45566
[info] play - Application started (Prod)
[info] play - Listening for HTTP on port 9000...
```

According to the Play Production documentation, this is what happens when you run the `start` command:

When you run the start command, Play forks a new JVM and runs the default Netty HTTP server. The standard output stream is redirected to the Play console, so you can monitor its status. If you type Ctrl-D, the Play console will quit, but the created server process will continue running in background. The forked JVM's standard output stream is then closed, and logging can be read from the *logs/application.log* file. If you type Ctrl-C, you will kill both JVMs: the Play console and the forked Play server.

Start command options

You can specify command-line options when issuing the `start` command. For example, the following command starts the server on port `8080`, while adjusting the minimum and maximum JVM heap size:

```
$ start -Dhttp.port=8080 -Xms512M -Xmx1G
```

There are also several ways to specify which configuration file to use. By default, Play uses the *application.conf* file it finds on the classpath, which by default is the *conf/application.conf* file from your application. You can specify a file on the local filesystem instead:

```
$ start -Dconfig.file=/myapp/conf/production.conf
```

The following command lets you load a *production.conf* file from the classpath:

```
$ start -Dconfig.resource=production.conf
```

If you keep that file in your application's *conf* directory, the Play `start` command will find it. Otherwise, place it on your application's classpath.

You can also load a configuration file from a URL:

```
$ start -Dconfig.url=http://foo.com/conf/production.conf
```

See the Play Configuration link in the See Also section for more options.

As noted in Recipe 1, you should never use the `run` command in production. According to the Play website, for each server request, a complete check is handled by SBT—not something you want to have happen in a production environment.

See Also

- Creating a standalone version of your application with `dist`
- Starting your application in production mode
- The Play Configuration page

17. Handling 404 and 500 Errors

Problem

You need to handle HTTP 404 and 500 errors in your application.

Solution

To handle 404 and 500 errors, create an object that extends the `GlobalSettings` trait, and override the necessary methods. To do this, create a file named *Global.scala* in your application's *app* directory with these contents:

```scala
import play.api._
import play.api.mvc._
import play.api.mvc.Results._

object Global extends GlobalSettings {

  // called when a route is found, but it was not possible to bind
  // the request parameters
  override def onBadRequest(request: RequestHeader, error: String) = {
    BadRequest("Bad Request: " + error)
  }

  // 500 - internal server error
  override def onError(request: RequestHeader, throwable: Throwable) = {
    InternalServerError(views.html.errors.onError(throwable))
  }

  // 404 - page not found error
  override def onHandlerNotFound(request: RequestHeader): Result = {
    NotFound(views.html.errors.onHandlerNotFound(request))
  }

}
```

The method `views.html.errors.onError(throwable)` refers to a Play template file I named *onError.scala.html* and placed in my *app/views/errors* folder:

```
@(throwable: Throwable)

@main("500 - Internal Server Error") {

  <h1>500 - Internal Server Error</h1>
  <p>@throwable.getMessage</p>

}
```

(Create the *app/views/errors* folder if it doesn't already exist.)

You can customize that code as desired, just like any other Play template.

The method `views.html.errors.onHandlerNotFound(request)` refers to a Play template file named *onHandlerNotFound.scala.html*, which is also in the *app/views/errors* folder. A simple version of that file looks like this:

```
@(request: RequestHeader)

@main("404 - Not Found") {

  <h1>404 - Not Found</h1>
  <p>You requested: @request.path</p>

}
```

Again, you can customize this template file as desired.

Discussion

As shown in the Application global settings page on the Play website, you can use this `Global` object for other purposes. For instance, the page demonstrates how to override the `onStart` and `onStop` methods of the `GlobalSettings` class to get a notice of when the application starts and stops:

```
import play.api._

object Global extends GlobalSettings {

  override def onStart(app: Application) {
    Logger.info("Application has started")
  }

  override def onStop(app: Application) {
    Logger.info("Application shutdown...")
  }

}
```

The Zentasks application that ships as a sample program with the Play distribution uses the `onStart` method to populate sample data for an application. You can find that application in the *samples/scala* directory of the Play distribution.

See Also

- Play application global settings
- The `GlobalSettings` trait

Index

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.