

Scala Pages

Author: Thomas Knierim

Software Version: 0.2.0

Document Version: 1.5, November 2010

What are Scala Pages?

Scala Pages (SCP) is a lightweight web framework for building Scala web applications. The framework itself is written in Scala. The author of this manual wrote SCP for his personal use. SCP is released under the Apache 2 license.

Motivation

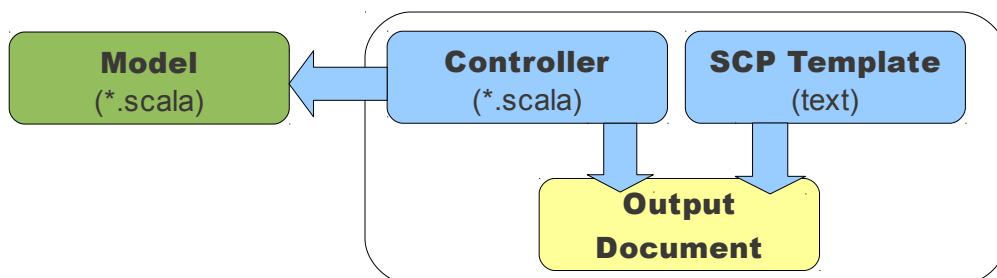
SCP was created with the following ideas in mind:

- to facilitate an MVC architecture
- to be light, fast, and scalable
- to provide an API that is easy to learn for Java programmers
- to make use of the request model provided by the Servlet API

SCP is built on and closely aligned to the Servlet API. It avoids reinventing the wheel and can be learned quickly by anyone familiar with that API.

Architecture

SCP is build around a text-oriented template engine. Dynamic web pages are either generated from templates or from Scala code. Templates can be HTML-templates, XML-templates, JSON-templates or any other text format, such as CSV, plain text, etc. Because SCP neither uses DOM nor SAX parsing, memory and CPU requirements are moderate. SCP templates make use of XML processing instructions instead of custom tags.



Features

- Designed for MVC
- Simple `$variable` syntax for template variables
- Simple `<?instruction ...?>` syntax for template processing
- Encoding/content detection, able to handle international text encodings
- Provides snippet interface for pluggable snippets (like custom tags)
- URL Rewriting
- Template Caching

Installation

Requirements

- Java Virtual Machine 5 or higher
- Scala 2.8 or higher
- Tomcat 6.x or higher
- Ant 1.7.x or higher

The installation package contains a project layout intended for use with Tomcat and Ant. However, you can replace Tomcat and Ant with the development environment of your choice. You only need to copy the scp.jar into your WEB-INF/lib.

Installation

Unzip the contents of scp.zip into any directory. Edit the ant build file build.xml and customise the file path settings in the first sections. In most cases, you only need to edit this section of the build.xml file:

```
<!-- root directory of this project -->
<property name="project.dir" value="."/>

<!-- root directory of Scala installation -->
<property name="scala.home" value="/usr/share/scala"/>

<!-- Tomcat installation directories -->
<property name="catalina.base" value="/var/lib/tomcat6"/>
<property name="catalina.home" value="/usr/share/tomcat6"/>

<!-- URL for Tomcat's manager application -->
<property name="catalina.manager.url" value="http://localhost/manager"/>

<!-- account name for Tomcat's admin account -->
<property name="catalina.manager.username" value="admin"/>

<!-- password name for Tomcat's admin account -->
<property name="catalina.manager.password" value="admin"/>
```

Deployment

Once you have adapted the path names in the build file to your local machine, you can deploy the “Hello world” sample application on your Tomcat server by typing “ant deploy” into a shell from the scp root directory. This should compile the sources, build a “hello-scp.war” file, and install it on your web server into the “hello-scp” context.

If this was successful, you can execute the “Hello World” application in your browser, e.g. <http://localhost:8080/hello-scp/>. Although the “Hello World” app is not particularly exciting, you can use it as a starting point for your own experimentation with SCP. Type “ant reload” into a shell after you make changes to the source. This should recompile and redeploy the application. Respectively, “ant build” only rebuilds source, while “ant war” rebuilds the war file, and “ant clean” deletes class files and all other assembled files.

Scala Pages Tutorial

MVC Implementation

Years of practice have shown that it is advantageous to separate program logic from presentation and SCP honours this principle by facilitating an MVC architecture. Dynamically generated pages are divided into two parts: the controller and the template. The controller responds to a page request, computes any required information and inserts the output into a variable container. The template is a text file (commonly HTML) that contains all of the presentation code. While there is usually a one-to-one correspondence between the controller and the template, there can also be multiple page templates, for example templates for different devices such as smart phones and tablets. If front controllers are used, there can also be multiple controllers. More about this later. When an output page is rendered, the SCP template processor inserts the information into the template and executes any processing instructions contained therein.

While the syntax elements of SCP templates are designed to harmonise with HTML and XML templates, the template processor itself is format-agnostic. It can process plain text files of any sort, including CSV, JSON, or an RFC 5322-formatted email template.

Here is an example of a controller that chooses a random colour from a list of colours:

```
import scala.util.Random
import scp.core._

class Page1(request: Request) extends TemplateResponse(request) {

  override def controller: Unit = {
    val colours = List(
      ("greenish", "#33cc66"),
      ("blueish", "#3399ff"),
      ("redish", "#cc3300"),
      ("shocking pink", "#ff00cc"),
      ("light purple", "#996699"),
      ("baby blue", "#66ffcc"),
      ("yellowish", "#fff33")
    )
    val pick = colours(Random.nextInt(6))
    request.vars += "nameOfColour" -> pick._1
    request.vars += "colour" -> pick._2
  }
}
```

This class extends the base class *TemplateResponse*, which is used for all dynamic pages with an associated template. It implements a single method named *controller*. The controller method picks a random colour and inserts the colour name and its HTML colour code into two request variables named *colour* and *nameOfColour*. The class definition “Page1” (Page1.scala) corresponds to a template file named *page1.scp*:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Colour Example</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  </head>
  <body>
```

```

    <h1 style="color:$colour">This ought to be $nameOfColour</h1>
  </body>
</html>

```

This template contains two variables: in the style attribute of the `<h1>` tag, which determines the text colour and in the text of the `<h1>` tag itself. These are replaced with the content defined by the controller.

Whenever the SCP framework executes a *TemplateResponse*, it loads a static template file from the file cache and merges its contents with the results of the calculation provided by the controller. SCP replaces the variables found in the template file and executes any processing instruction embedded in the template file (there is none in the above example). This is done in a single linear parsing step.

The linking piece of information is provided by:

```

request.vars += "nameOfColour" -> pick._1
request.vars += "colour" -> pick._2

```

These instructions define two request scope variables named *nameOfColour* and *colour* that take on the String values of the randomly picked tuple. We could also have written it in a more traditional Java-like notation:

```

request.vars.add("nameOfColour", pick._1)
request.vars.add("colour", pick._2)

```

This does exactly the same thing. All request variables are of type *Any*. The template always renders the standard string representation of variables.

In addition to request variables, which are forgotten after the response has been sent to the client, there are session scope variables and application scope variables. Session scope variables are associated with a user session that is maintained by the web container. They provide an easy way to preserve state across a series of requests. Application scope variables are global to the web application. They are visible for all users. Since application scope variables pose concurrency and security challenges, it's best to use them only in special cases, or not use them at all. Session and application variables are set in a similar way:

```

session.vars += "user" -> "thomas"
session.vars += "isLoggedIn" -> true
app.vars.add("someObject", someObject)

```

Variables can be removed as follows (there are again scala-ish and java-ish variants):

```

session.vars -= "user"
session.vars.remove("isLoggedIn")

```

Reading a variable requires a bit more code, because we need to do type checking:

```

val isLoggedIn = session.vars("isLoggedIn") match {
  case Some(b: Boolean) => b
  case _ => false
}

```

However, in most cases, variables are simply set and passed through to the template processor. Therefore, the advantage of convenient writes outweighs convenient reads.

Besides functioning as variable containers, the *request*, *session*, and *app* objects also provide methods for accessing parameters, headers, cookies and more. They are basically wrappers for the corresponding `HttpServletRequest`, `HttpSession`, and `ServletContext` interfaces of the Java Servlet API. Therefore, the SCP API may seem familiar to Java web programmers. For example, to retrieve a request header we simply write:

```
val contentLength = request.header("Content-Length")
```

To retrieve a list of all request headers (names and values), we write something like this:

```
val headers = request.headerNames.map(name =>
  (name, request.header(name).get))
```

A request parameter is retrieved as follows:

```
val amount = request.param("amount")
```

This gives us an `Option(String)` value, so we need to check whether the parameter was actually set. Alternatively, we can use this extended form to specify a default value:

```
val amount = request.param("amount").getOrElse("")
```

This returns the request parameter *amount* as a string, or the second function argument (“empty string”) as default if the parameter was not set. If we know that the request parameter should be of a certain type, we can call a corresponding method that converts the value:

```
val amount = request.paramAsInt("amount")
```

This gives us an `Option(Int)` value with safe conversion logic, which means if the request parameter was not set, or if it is not an integer, we get `None`. There are methods for every different `AnyVal` type, such as `Boolean`, `Float`, etc. Again, the method calls can be extended to provide default values, for example:

```
val amount = request.paramAsInt("amount").getOrElse(122)
val taxes = request.paramAsFloat("taxes").getOrElse(7.5)
```

Here is a another code example named `Page2.scala`:

```
import scp.core._

class Page2(request: Request) extends TemplateResponse(request) {

  def controller: Unit = {
    request.vars.add("myColor", "green")
    val xmlString = "<b>This ought to be bold.</b>"
    request.vars.add("xmlVar", xmlString);
    request.vars.add("someVar", xmlString);
    request.vars.add("someList", List(1,2,3,4,5))
    session.vars.add("spices", Array("Pepper", "Salt", "Curry", "Oregano"))
    request.vars.add("capitals", Map("Italy" -> "Rome", "Germany" -> "Berlin",
      "France" -> "Paris", "Spain" -> "Madrid", "Norway" -> "Oslo"))
  }
}
```

This controller defines request and session variables with commonly used Scala types, such as scalar variables, a list, an array and a map. The corresponding template `page2.scp` shows how these variables are used in the template:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>SCP Variables</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  </head>
  <body>
    <h2 style="color:$myColor">Welcome to Scala Pages!</h2>
    <!-- Variables are normally xml-escaped -->
    This variable is XML-escaped: $someVar<br/>
    <!-- Unless their name begins with "xml" -->
    This variable is not XML-escaped: $xmlVar<br/>
    This $$variable is not a variable.<br/>
    <!-- Let's look at how a simple list is printed -->
    The $$someList request variable contains: ${someList}.<br/>
    <!-- Now let's try conditional output -->
    <?scp-echo if="$someList.isEmpty" value="This list is empty.<br/>"?>
    <?scp-echo if="!$someList.isEmpty"
      value="It has $someList.length elements.<br/>"?>
    Here we have some delicious spices:<br/>
    <!-- Now print the contents of an array using iteration -->
    <ul><?scp-echo value="<li>$spice</li>" iterate="$session.spices"
      as="$spice"?></ul>
    <!-- We can pick any element of an ordered sequence like this: -->
    My favourite spice is ${session.spices(2)}.<br/>
    <!-- We can pick a map value by its key using dot notation -->
    The capital of Spain is ${capitals.Spain}.<br/>
    <!-- Or we can iterate and print the values of a map -->
    <ul><?scp-echo iterate="$capitals.values"
      as="$city" value="<li>$city</li>"?></ul>
    Now let's look at some predefined variables:<br/><br/>
    url=$url<br/>
    rooturl=$rooturl<br/>
    querystring=$querystring<br/>
  </body>
</html>

```

In case you are wondering about the unorthodox comment notation:

```
<!--- This comment does *NOT* appear in the output. -->
```

The template processor deletes comment lines with three dashes from the output, while it leaves regular XML comments with two just dashes intact.

```
<!-- This comment is visible in the output. -->
```

Unclosed comments produce a parsing error, even in non-XML template files. Another hint in this example is that the dollar sign can be printed as a literal simply by escaping it with a second dollar sign:

```
Item price: $$20.-
```

If you load this page into your browser, you should see something like this:

Welcome to Scala Pages!

This variable is XML-escaped: This ought to be bold.

This variable is not XML-escaped: **This ought to be bold.**

This \$variable is not a variable.

The \$someList request variable contains: List(1, 2, 3, 4, 5).

It has 5 elements.

Here we have some delicious spices:

- Pepper
- Salt
- Curry
- Oregano

My favourite spice is Curry.

The capital of Spain is Madrid.

- Madrid
- Rome
- Oslo
- Paris
- Berlin

Now let's look at some predefined variables:

url=/chronolog/page.xhtml

rooturl=/chronolog

querystring=

Response Type

The SCP framework offers several choices for implementing request handlers/controllers embodied by different Response classes. Each of these response classes can be extended by the application:

Class Name	Purpose
Response	Low-level handling using Servlet API
TextResponse	Create a plain text response using Scala code
XmlResponse	Create a XML/HTML response using Scala code
TemplateResponse	Create a response using code (controller) and a template (view)

TemplateResponse

The *TemplateResponse* class is probably the most useful choice for an MVC application. We have already seen code examples that demonstrate its use. In addition to the abstract *controller()* method, *TemplateResponse* provides several state fields that can be utilised by the web application:

Name	Type	Purpose
var contentType	String	Determines the MIME content type of the response. By default this is either detected from the template or set to the application-wide default.
var encoding	String	Determines the character encoding of the response. By default this is either detected from the template or set to the application-wide default.
var variableEscaping	String	Determines how variables are escaped for output by the template processor (e.g. XML HTML, Javascript, CSV or none). Defaults to the application-wide default setting.
var templateFileName	String	Determines the template to be loaded. By default the template file name is derived from the request URL String.

TextResponse

The `TextResponse` class is appropriate for generating plain text output. It provides an abstract `text()` method that must be implemented to return the response as a `CharSequence`, i.e. a `String`, `CharBuffer`, `StringBuffer`, or `StringBuilder`. The following example creates CSV output containing all system properties:

```
import scp.core._
import scp.util.Str

class Page3(request: Request) extends TextResponse(request) {

  contentType = "text/csv"
  addHeader("Content-Disposition", "attachment; filename=properties.csv")

  def text: StringBuilder = {
    val result = new StringBuilder()
    val propertyNames = System.getProperties().propertyNames()
    while (propertyNames.hasMoreElements) {
      val key = propertyNames.nextElement.asInstanceOf[String]
      val value = System.getProperty(key)
      if (value != null) {
        result.append(Str.escapeCSV(key))
        result.append(",")
        result.append(Str.escapeCSV(value))
        result.append("\n")
      }
    }
    result
  }
}
```

This short piece of code creates a CSV file for download containing all system environment properties of the server. Not recommended for online systems – just for illustration. The assignment to `contentType` in the constructor defines the output MIME type and the `Content-Disposition` header forces a download in the browser.

TextResponse provides two state fields that can be utilised by the web application:

Name	Type	Purpose
var contentType	String	Determines the MIME content type of the response. This defaults to the application-wide settings.
var encoding	String	Determines the character encoding of the response. This defaults to the application-wide settings.

XMLResponse

The *XMLResponse* class is most useful for generating XML output entirely in code. The following code demonstrates the canonical Hello World example:

```
class Page4(request: Request) extends XmlResponse(request) {

  val title = "Welcome to SCP"
  val name = "World"

  def xml =
    <html>
      <head>
        <title>{title}</title>
      </head>
      <body>
        <h1>Hello, {name}!</h1>
      </body>
    </html>
}
```

This may not be tremendously useful, but it illustrates the basics of an *XMLResponse*. The application must implement the abstract method *xml()* to return an object of type *scala.xml.NodeSeq*. Another example shows how this may be put to use in a real world application to generate an ATOM XML feed from blog postings:

```
class Page5(request: Request) extends XmlResponse(request) {

  contentType = "application/atom+xml"
  val topic = request.param("topic").orElse("Some Other Topic")
  val year = request.paramAsInt("year").orElse(2010)
  val posts = getPostsFor(topic, year)

  def xml =
    <feed xmlns="http://www.w3.org/2005/Atom">
      <title>BlogTitle</title>
      <subtitle>Something very interesting.</subtitle>
      <link href="http://myblog.com/" />
      <link href="http://myblog.com/atom.xml" rel="self" />
      <updated>{posts(0).atomDate}</updated>
      <author>
        <name>John Smith</name>
        <uri>http://myblog.com/about.html</uri>
      </author>
    </feed>
```

```

</author>
<id>http://myblog.com/</id>
{for (post <- posts) yield
<entry>
  <title>{post.title}</title>
  <link href={"http://myblog.com/" + post.slug + ".html"} rel="alternate"/>
  <id>{post.atomId}</id>
  <updated>{post.atomDate}</updated>
  <content type="html">{post.body}</content>
  <author>
    <name>John Smith</name>
    <uri>http://myblog.com/about.html</uri>
  </author>
</entry>
}
</feed>
}

```

XMLResponse provides two state fields that can be utilised by the web application:

Name	Type	Purpose
var contentType	String	Determines the MIME content type of the response. This defaults to the application-wide settings.
var encoding	String	Determines the character encoding of the response. This defaults to the application-wide settings.

Response

All response classes extend from a single base class named *Response*. You can extend from this class directly in order to create a low-level custom response. In this case, you have to utilize the Servlet API. This is only necessary if you require special request processing, for example, if you want to implement different methods for GET and POST requests or for other types of HTTP requests, or if you create a data upload handler. Implementing a request handler by extending from *Response* directly is essentially the same as implementing a servlet. The following example measures the size (in bytes) and the time (in ms) of a data stream uploaded via the POST method:

```

class Page6(request: Request) extends Response(request) {

  val req = request.httpServletRequest
  val rsp = request.httpServletResponse
  rsp.setContentType("text/plain");
  rsp.setStatus(HttpServletResponse.SC_OK);

  override final def doPost(): Unit = {
    // Upload - read data from POST request
    val buffer = new Array[Byte](1000)
    var elapsedTime = 0L
    var transmittedBytes = 0L
    var numberOfBytes = 0
    val clockTime = System.currentTimeMillis()
    val in = req.getInputStream()
    while (numberOfBytes != -1) {
      numberOfBytes = in.read(buffer)
    }
  }
}

```

```

    if (numberOfBytes != -1)
      transmittedBytes += numberOfBytes;
  }
  elapsedTime = System.currentTimeMillis() - clockTime;
  val out = rsp.getOutputStream();
  out.write((transmittedBytes.toString + " Bytes\r\n").getBytes());
  out.write((elapsedTime.toString + " ms\r\n").getBytes());
  out.close();
}
}

```

Response provides all of the standard servlet service methods which may be overridden by the application:

Name	Purpose
def doGet()	Handler for GET requests only
def doPost()	Handler for PUT requests only
def doDelete()	Handler for DELETE requests only
def doHead()	Handler for HEAD requests only
def doOptions()	Handler for OPTIONS requests only
def doPut()	Handler for PUT requests only
def doTrace()	Handler for TRACE requests only

In addition, *Response* provides methods that can be utilised by the application. Since *TextResponse*, *XmlResponse*, and *TemplateResponse* extend from *Response*, these are available with all response types:

Name	Purpose
def app	Returns application object
def session	Returns session object
def setHeader()	Sets the value of an HTTP header in the response
def addHeader()	Adds an HTTP header to the response
def containsHeader()	Checks whether response contains a certain header
def addCookie()	Adds a cookie to the response
def resolve()	Resolver for converting an URI to a file name

URI Resolution

At this point you may wonder how the examples are invoked in a browser and how browser URIs are resolved to SCP templates and response classes. The answer is by package name and class name. You can't really tell from the above examples, because the examples don't contain package declarations. The package name is implicit in this case, because a source file without package declaration automatically lands in the default package, which in terms of URI resolution means the context path (= root URL). Thus a class with the name "Test" that is deployed in a web context named "scp" corresponds to the URI /scp/test.scp. If you are not comfortable with mapping the extension ".scp" to SCP web pages, you can change it in the SCP configuration file. More about

this later.

By default, browser URIs are resolved to request handler class names in an intuitive fashion. For example, the response class “Display.scala” which is part of the package `com.myproject.catalog` corresponds to the URL `/$rooturl/com/myproject/catalog/display.scp`, whereas `$rooturl` is the context root directory. Thus there is a one-to-one correspondence between URIs and package names. It is possible to shorten URIs, however. In the named example, the part `com.myproject` is the same for all response classes and doesn't need to be included in the URI. To shorten URIs, you can define the root package for response classes in the SCP configuration file `config.properties`. If you set the property `handlerPackagePrefix` to “`com.myproject`”, this part of the URI can be omitted. The resulting URI for the named response class would then be reduced to `$rooturl/catalog/display.scp`.

URIs are resolved according to these rules:

- The part corresponding to the package name is converted to lower case and dashes are replaced by underscores
- The part corresponding to the class name (or file name) is converted from underscores or dashes to CamelCase, thus `product.scp` becomes `Product`, `special_offers.scp` becomes `SpecialOffers` and `check-out-now.scp` becomes `CheckOutNow`.

If you are not comfortable with the standard URL resolution, you can define URL rewriting rules, similar to Apache HTTP. This feature also makes it possible, to create search-engine-friendly URIs for SCP pages with complicated query strings.

Template files are likewise resolved according to the request URI. A request with the URI `$rooturl/catalog/display.scp` will by default look for a template with the name “`display.scp`” in the `/catalog` directory of the web document root directory. This default behaviour can be altered by setting the `templateFileName` field of a `TemplateResponse` handler, or by overriding the `resolve()` method. Includes in templates are resolved relative to the template directory.

SCP Variables

The code examples we have seen illustrate the basic use of template variables. An SCP variable always starts with a “\$” dollar sign and consists of a sequence of alphanumerical characters including the underscore, for example `$foo` or `$foo_bar`. It must begin with a letter. SCP variables reference Scala objects, fields, or parameterless methods/functions of any type. A variable expression is string that begins with with a variable name and contains one or more “.” dots to dereference fields. It may also contain “()” parentheses to dereference an element of an ordered sequence, such as a list. An example of a variable expression is `$session.mySpices(3)`.

The notation for variables is as simple as it can be. Usually, the dollar sign prefix is sufficient to denote a variable. Curly braces can optionally be placed around the variable expression, for example: `#{myList}`. This is necessary if the variable name contains blanks, or if the variable expression needs to be separated from the surrounding text, for example, if the variable expression stands before a “.” dot character in the text. The literal dollar sign can be inserted into the text simply by repeating it: `$$`.

SCP inserts the `toString()` representation of the referenced variable into the output text. All values are escaped according to the default escaping setting (ordinarily XML escaping). If the variable name begins with “`xml`”, XML escaping is suppressed. If the variable name begins with “`html`”, HTML escaping is suppressed. Here are some examples of variables and non-variables:

```

$abc10          // is a variable
$üéßÅ          // is a variable
${var iabl e}  // is a variable
$$abc          // is NOT a variable
$50.-          // is NOT a variable
$10abc         // is NOT a variable
$ abc          // is NOT a variable
$(jquery)      // is NOT a variable

```

SCP uses familiar scope identifier prefixes:

```

$request.foo    // refers to a REQUEST variable named "foo", same as $foo
$session.foo    // refers to a SESSION variable named "foo"
$app.foo        // refers to an APPLICATION variable named "foo"
$params.foo     // refers to a REQUEST PARAMETER named "foo"

```

In addition, SCP provides several predefined variables:

```

$rooturl        // webapp context (=request.getContextPath())
$resource       // filename of the resource (e.g. "foo.scp")
$url            // request URI (=request.getRequestURI())
$qualifiedurl   // qualified URI (=request.getRequestURL())
$querystring    // the query string (=request.getQueryString())
$server         // server name (=request.getServerName())
$scheme         // http, https or ftp (=request.getScheme())
$port           // port number (=request.getServerPort())
$protocol       // protocol, e.g. HTTP/1.1 (=request.getProtocol())

```

Character Encoding And Content Type Detection

SCP templates are text files that can be stored using any character encoding. The encoding may be specified in the template itself. When parsing a template file, SCP looks for either of these three string patterns to determine encoding:

- (1) XML encoding declaration: `<?xml [...] encoding="encoding" [...] ?>`
- (2) HTML declaration: `<meta [...] charset="encoding" [...] \>`
- (3) SCP style (proprietary): `<?scp-content [...] encoding="encoding" [...] ?>`

Enclosing double quotation marks for attributes are optional. If any of these patterns is encountered in the first 4096 characters of the template, the file content is read using the given encoding and the HTTP response is set to the the same encoding. Likewise, SCP attempts to determine the content type by looking for these patterns:

- (1) HTML style: `<meta [...] content="content-type" [...] />`
- (2) SCP style: `<?scp-content [...] type="content-type" [...] ?>`

Again, enclosing double quotation marks for attributes are optional. If any of these patterns is encountered in the first 4096 characters of the template, SCP sets the HTTP response headers to the given MIME type. Character encoding and content type declarations must not contain variables and they must be specified in the main template. Declarations in include files are presently not recognised.

SCP Processing Instructions

SCP makes use of XML processing instructions for certain functionality instead of XML custom

tags. That way, SCP templates can be edited with your favourite HTML/XML editor without having to teach your editor new tricks and no DTD or schema definition is required. SCP processing instructions are also used for non-XML text templates, in which case they can simply be thought of as text markers. Processing instructions are always replaced with their corresponding output. You have already seen one such processing instruction in the previous paragraph.

PI	Description	Example
<code><?scp-content ...?></code>	Specifies template content type and character encoding.	<code><?scp-content type="text/html" encoding="UTF-8"?></code>
<code><?scp-include ...?></code>	Includes another template file.	<code><?scp-include src="../footer.scp"?></code>
<code><?scp-echo ...?></code>	Prints text and variables.	<code><?scp-echo value="\$someVariable"?></code>
<code><?scp-number ...?></code>	Prints a formatted number using <code>java.text.DecimalFormat</code>	<code><?scp-number value="\$num" format="#,###.##"?></code>
<code><?scp-date ...?></code>	Prints a formatted date using <code>java.text.SimpleDateFormat</code>	<code><?scp-date value="\$today" format="EEE MM dd yyyy"?></code>
<code><?scp-snippet ...?></code>	Invokes a snippet class and inserts its output.	<code><?scp-snippet class="mySnippetClass"?></code>

Number values must be of a numeric type, or a numeric String. Date values must be of type `Long` or `java.util.Date`.

The most interesting processing instructions are probably `<?scp-include ...?>` and `<?scp-snippet ...?>`. The include instruction includes the parsed contents of the specified source file at the text position of the instruction. The `src` parameter is either a relative or absolute path, or an external URI. Includes are not checked for recursion, so if you have two templates including each other, this would most likely result in a stack overflow.

Conditional Statements

SCP processing instructions can optionally include an *if* attribute which decides whether the instruction is executed or not. This construct is deliberately kept simple to prevent application logic from creeping into templates. The only thing that the *if* clause does is distinguishing logical true from logical false values.

```
<p>Schrodinger's cat is
  <?scp-echo value="alive." if="$alive"?>
  <?scp-echo value="dead." if="!$alive"?>
</p>
```

The above two lines produce the sentence “Schrodinger's cat is dead.” or “Schrodinger's cat is alive.” depending on the value of the *\$alive* variable. The boolean result of different types is interpreted according to the following scheme: if the variable is null, the result is false. If the variable is Boolean, the results takes on the boolean value. If the variable is of type *Option*, it is false if *None*, true otherwise. If the value is of type *Number*, it is false if it equals zero, true otherwise. If the value is of any other type, it is false if its String representation (`toString`) results in an empty String, true otherwise. The “!” exclamation mark can be used to negate the result. There are no logical operators and no comparison operators.

Iteration

All SCP processing instructions may be iterated by combining them with an *iterate* argument:

```
<?scp-echo value="-" iterate="50"?>
```

This prints out a line consisting of 50 dashes.

```
<?scp-include src="fragment.html" iterate="$iterator"?>
```

This includes and parses the file *fragment.html* for as many times as specified by the variable *\$iterator*. If *\$iterator* is an integer variable of value *n*, the include operation is repeated *n* times. If *\$iterator* is an object of type *Iterable* or an array, the operation is repeated as many times as there are elements in the iterable object, and upon each iteration the variable name *\$iterator* is set to the value of the element.

```
<?scp-echo value="Element: $collection\n" iterate="$collection"?>
```

This instruction prints all elements of *\$collection*, each on a separate line. The variable name *\$collection* is reused for the loop variable. If you would like to use a different variable name instead, you can specify the *as* argument:

```
<?scp-echo value="Element: $element\n" iterate="$collection" as="$element"?>
```

This does exactly the same as the previous instruction, except that it uses a specific loop variable name which hopefully makes the instruction more readable. You can also use this instruction to output short sequences of XML/HTML:

```
<ul><?scp-echo value="<li>$element</li>\n" iterate="$collection" as="$element"?></ul>
```

The above example renders the elements of the collection as an HTML list with bullets. Since according to XML syntax, angle brackets and other entity references are allowed in processing instructions (unlike in regular XML elements), this provides a convenient way to insert short XML sequences into the output text, for example by filling a list or a table row with values. If on the other hand, the XML sequence is more complex, it may be better to use a code snippet instead of a literal sequence:

```
<ol><?scp-snippet class="MySnippet" iterate="$collection" as="$element"?></ol>
```

Snippets

A snippet is a reusable Scala class that produces some output depending on the parameters passed to it. Snippets receive all arguments of the processing instruction as input parameters. There are two main uses for snippets: (1) To encapsulate complex display logic and present a simple processing instruction syntax for convenient use in templates. This usage is similar to JSP custom tags, although SCP snippets are much easier to implement. (2) To encapsulate application logic and thus prevent it from being coded in the template. Typical examples for this would be complex conditional execution or data filtering. Snippets can be understood as a construction kit for extending SCP processing instructions.

Here is a very simple example. The following processing instruction outputs text in boldface:

```
<?scp-snippet class="BoldText" text="This text is printed bold."?>
```

...Or

```
<?scp-bold-text text="This text is printed bold."?>
```

The respective snippet class looks like this:

```
import scp.core._

class BoldText extends Snippet {

  override def render(args: Map[String, Any]): String =
    "<b>" + Str.escapeXML(args("text").toString) + "</b>"
}
```

Several things should be noted:

- This example is just for illustration. Of course, one doesn't really need to create a snippet for printing bold text.
- The snippet class must implement a method named *render()*.
- The *render()* method takes a *Map* as an argument. This map contains all attributes of the processing instruction, indexed by attribute name. Because the values of this map are of type *Any*, the snippet is responsible for converting them to the proper data type.
- The *render()* method returns a string that is inserted directly into the output. The snippet is responsible for escaping the output text.
- The processing instruction can be shortened by incorporating the class name into the name of the processing instruction.

A snippet class can be part of any package and can be referenced by its fully qualified class name:

```
<?scp-snippet class="com.myproject.mysnippets.BoldText" text="This text is printed bold."?>
```

Since this can be cumbersome to write, it is possible to define a standard package name for all snippets in the SCP configuration file. With this definition in place, the package prefix can be omitted:

```
<?scp-snippet class="BoldText" text="This text is printed bold."?>
```

Furthermore, the class name can be contracted into a processing instruction name that begins with "scp-" plus the class name, except that camel case notation is replaced by word separation using dashes:

```
<?scp-bold-text text="This text is printed bold."?>
```

It's important to remember that snippet arguments are defined implicitly and (at this time) aren't type safe. In the above example, this does not constitute a problem, because every object of type *Any* has a *toString()* method. Generally speaking, you are on the safe side as long as you deal with text. However, the template may pass on data objects of any type to the snippet. Say, for example, the controller defines a list of lists named *datatable* to output tabular data. The snippet processing instruction passes on the *datatable* variable simply by inserting the variable into the attribute value:

```
<?scp-snippet class="ShowTable" datatable="$datatable"?>
```

In this case, the *render()* method of the snippet receives a map with a single element, whose key is "datatable" and whose value is of type *List* with elements of of type *List*. The snippet class

implementation should make sure, that the expected type is present. Let's illustrate this with a more sophisticated example. We are going to write a snippet class that takes an iterable series of data, such as a list, and prints it out in a tabular form with a given number of columns and with specified formatting. We use an HTML table element for this purpose. The data can be arranged horizontally (across rows) or vertically (across columns).

```
import scp.core._
import scp.util._

/**
 * Arranges a list of items in a tabular format.
 *
 * Supported attributes:
 *
 * items = contains a collection with output data
 * cols = number of columns in which to display items (default = 1)
 * arrange = vertical or horizontal (default = horizontal)
 * class = CSS class for table cells (default = none)
 * style = CSS inline style for table cells (default = none)
 * tableclass = CSS class for table element (default = none)
 * tablestyle = CSS inline style for table element (default = none)
 */

class TabularList extends Snippet {

  override def render(args: Map[String, Any]): String = {
    val list = args.get("items") match {
      case Some(x) => x
      case None => List()
    }
    if (list.isInstanceOf[Iterable[_]]) {
      val xmlEscaped = list.asInstanceOf[Iterable[_]].map(x =>
Str.escapeXML(x.toString))
      if (!xmlEscaped.isEmpty) genHtml(xmlEscaped, args) else ""
    }
    else ""
  }

  def genHtml(iterable: Iterable[Any], args: Map[String, Any]): String = {
    val array = iterable.toArray
    val length = array.length
    val cols = try {
      args("cols").toString.toInt
    }
    catch {
      case e: Exception => 1
    }
    val rows = (length / cols) + (if (length % cols == 0) 0 else 1)
    val isVertical = args.get("arrange") match {
      case Some(x) => x.toString.equals("vertical")
      case None => false
    }
    val isClassDefined = args.get("class").isDefined
    val clazz = if (isClassDefined) args("class").toString else ""
    val isStyleDefined = args.get("style").isDefined
    val style = if (isStyleDefined) args("style").toString else ""
    val isTableClassDefined = args.get("tableclass").isDefined
    val tableClass = if (isTableClassDefined) args("tableclass").toString else

```

```

"""
    val isTableStyleDefined = args.get("tablestyle").isDefined
    val tableStyle = if (isTableStyleDefined) args("tablestyle").toString else
"""
    // begin output
    val html = new StringBuilder
    if (length > 0) {
        html.append("<table")
        if (isTableClassDefined) {
            html.append(" class=\"")
            html.append(tableClass)
            html.append('\"')
        }
        if (isTableStyleDefined) {
            html.append(" style=\"")
            html.append(tableStyle)
            html.append('\"')
        }
        html.append(">\n")
        for (i <- 0 until rows) {
            html.append(" <tr>")
            for (j <- 0 until cols) {
                val index = if (isVertical) j * rows + i else i * cols + j
                val data = if (index < length) array(index) else ""
                html.append("<td")
                if (isClassDefined) {
                    html.append(" class=\"")
                    html.append(clazz)
                    html.append('\"')
                }
                if (isStyleDefined) {
                    html.append(" style=\"")
                    html.append(style)
                    html.append('\"')
                }
                html.append('>')
                html.append(data)
                html.append("</td>")
            }
            html.append("</tr>\n")
        }
        html.append("</table>\n")
    }
    html.toString
}
}

```

For the sake of reusability, the *items* attribute is of type *Iterable*, which means it can be used with any type of collection. Let's assume the handler defines a variable *\$mySpices*:

```

val mySpices = Array("anise", "basil", "curry", "dill", "fennel", "ginger",
"mint", "nutmeg", "oregano", "pepper", "rosemary", "salt", "thyme", "vanilla")
request.vars += ("mySpices", mySpices)

```

The following SCP processing instruction renders the items of this list in a table with three columns, each 100 pixel wide:

```

<?scp-tabular-list items="$mySpices" cols="3" arrange="vertical"
style="width:100px;"?>

```

The result roughly looks like this in a browser:

anise	ginger	rosemary
basil	mint	salt
curry	nutmeg	thyme
dill	oregano	vanilla
fennel	pepper	

Extending Snippets

Because Scala is object-oriented, snippet classes are well-suited for extension. Building a library of template processing instructions is therefore quite straightforward. The guiding principle is to implement basic functionality as abstract classes and/or traits extending from *Snippet*. These building blocks can then be enhanced through inheritance. For example, we can build on the above snippet and define another snippet class that displays an option group, i.e. a group of radio buttons, arranged in a table.

```
import scp.core._
import scp.util._

/** Arranges a group of options (radio buttons) in tabular format.
 *
 * Supported attributes:
 *
 * name: name of the HTML radio input element (required)
 * options: contains either a list or a map of String options (required)
 * sel: preselected option (default = none)
 *
 */
class TabularOptions extends TabularList {

  override def render(args: Map[String, Any]): String = {
    var result = ""
    val html = new StringBuilder
    val name = args("name").toString
    val isSelected = args.get("sel").isDefined
    val selectedItem = if (isSelected) args("sel").toString else ""
    val inputElement = "<input type=\"radio\" name=\"" + Str.escapeXML(name) +
"\\" value=\""
    val list = new scala.collection.mutable.ListBuffer[String]
    if (args("options").isInstanceOf[Map[_,_]]) {
      val options = args("options").asInstanceOf[Map[String, String]]
      for ((key, value) <- options) {
        html.setLength(0)
        html.append(inputElement)
        html.append(Str.escapeXML(key))
        if (isSelected && selectedItem.equals(key))
          html.append("\" checked=\"checked\"/>")
        else
          html.append("\"/>")
        html.append(Str.escapeXML(value))
        list += html.toString
      }
    }
    result = genHtml(list, args)
  }
}
```

```

}
else if (args("options").isInstanceOf[Iterable[_]]) {
  val options = args("options").asInstanceOf[Iterable[String]]
  for (value <- options) {
    html.setLength(0)
    html.append(inputElement)
    val escapedValue = Str.escapeXML(value)
    html.append(escapedValue)
    if (isSelected && selectedItem.equals(value))
      html.append("\" checked=\"checked\"/>")
    else
      html.append("\"/>")
    html.append(Str.escapeXML(escapedValue))
    list += html.toString
  }
  result = genHtml(list, args)
}
result
}
}

```

In addition to the inherited style attributes named *cols*, *arrange*, *class*, *style*, *tableclass*, and *tablestyle*, this snippet class supports the attributes *name*, *options*, and *sel*. The options argument is either a *Collection*, in which case the HTML values and display names of the radio buttons are identical, or a *Map* in which case the key is used for the HTML value attribute and the value is used for the HTML display name.

Configuration

The SCP web framework has few configuration settings. All settings are stored in a single properties file: WEB-INF/config.properties. The file format complies with the standard Java .properties format as defined in java.util.properties. Alternatively, SCP configuration settings may be specified directly as servlet parameters in the web.xml deployment descriptor.

Property Name	Description
defaultEncoding	Default character encoding for dynamic web pages (e.g. ISO-8859-1). Is set to "UTF-8" if not specified.
defaultContentType	Default content type (MIME type) setting for dynamic web pages (e.g. text/html). Is set to "text/html" if not specified.
defaultVariableEscaping	Specifies how template variables are escaped in the output (default="xml"). "none" = no escaping "xml" = XML escaping "html" = HTML escaping "csv" = CSV escaping "javascript" = JavaScript escaping
ignoreTemplateVariableErrors	Specifies whether an undefined variable in a template causes an exception or not (default=false).
virtualWelcomeFile	Specifies the name of a virtual welcome file for directory URLs where the web directory doesn't contain a physical index.* file. This property must be specified to map URLs with a trailing "/" to an "index" request handler (default=index.scp).

handlerPackagePrefix	Specifies the prefix of the package where the SCP framework looks for response classes (=handlers). Example: If handlerPackagePrefix is set to "com.mydomain.handlers" and the request URL is "\$rooturl/products/overview.scp", the framework will look for a request handler class named "Overview" in the package "com.mydomain.handlers.products" (default=empty).
snippetPackagePrefix	Specifies the prefix of the package where the framework looks for SCP snippet classes. This string is prepended to the "class" attribute of the <?scp-snippet ?> processing instruction.
rewriteRules	Specifies rules for URL rewriting. Each rule is comprised of two strings separated by one or more blanks. The first string contains a regular expression that is used for URL matching. The second string contains the rewritten URL. Use \$1, \$2, \$3, etc. to reference regular expression groups in their given order. Individual rewrite rules are separated by comma.
frontControllers	Specifies one or more front controllers. Each front controller is specified by a pair of strings separated by one or more blanks. The first string contains a regular expression that is used for URL matching. The second string contains the qualified class name of the front controller. Multiple front controllers are separated by comma.

Configuration data in the config.properties is accessed by the framework through the *Config* class. This class can also be used by the application to access user-defined properties in the config.properties file or in the web.xml deployment descriptor. Config defines a number of *get()* methods for retrieving configuration data.

Front Controllers

Occasionally, it is necessary to implement controllers not just for a single request URL, but for a group of requests, or for all requests of an application. For example, you might want to authenticate users in order to access certain parts of the web application. Or you might want to generate logging or tracking information for web resources. Or you might want to compute certain values to show on every page of a web application. The SCP web framework provides a front controller mechanism for these purposes. It is based on the javax Filter API.

Front controllers are similar to regular SCP controllers. A front controller is a Scala object that extends from the *FrontController* class. The front controller class provides two methods *before()* and *after()* which are invoked by the framework. The *before()* method executes before any response class is called, and even before URL rewriting is performed. It returns a boolean value that indicates whether request processing should be continued (true) or intercepted (false). The *after()* method is invoked after response class processing and page rendering is completed.

Since front controllers are based on filters, they are by default executed not just for dynamic page requests, but for every request, including requests for static resources, such as images or HTML files. This behaviour can be changed by adjusting the filter configuration in the *web.xml* descriptor appropriately. Front controllers can either be instantiated at application startup (preloaded) or at each request. It is important to note that preloaded front controllers can share data fields across requests.

The following front controller example implements a simple access logger, similar to that of your web server, keeping track of access time, IP, URI and the time needed to serve a page.

```
import scp.core._

class AccessLogger(app: WebApp) extends FrontController(app) {

  def before(request: Request): Boolean = {
    val timestamp = System.currentTimeMillis()
    request.vars += "timestamp" -> timestamp
    true
  }

  def after(request: Request): Unit = {
    val timestamp = request.vars("timestamp") match {
      case Some(x: Long) => x
      case _ => 0L
    }
    var duration = System.currentTimeMillis() - timestamp
    val output = (new java.util.Date()).toString +
      " === " +
      request.qualifiedURL +
      " === " +
      request.clientIP +
      " === " +
      duration + " ms"
    scp.util.IO.stringToFile(output + "\n", new java.io.File("/tmp/web.log"),
true)
  }
}
```

The time needed to serve a web page is computed by the difference of the system clock value when entering the *before()* and *after()* methods. Please note that *timestamp* is stored in a request variable. Implementing it as a field would make the *AccessLogger* class non-reentrant and would not work in a multi-user environment. The *after()* method appends a single line for each request to the */tmp/web.log* file. In order to deploy the *AccessLogger* class in a web application, it must be added to the *config.properties* configuration file. Each front controller is associated with a URL, or rather a URL pattern. This is done by specifying a regular expression pattern in the *config.properties* file:

```
frontControllers=.*\.scp(\?.*)? myapp.frontcontrollers.AccessLogger preload
```

We assume that the *AccessLogger* class is part of the package *myapp.frontcontrollers*. The expression on the right side of the = sign contains two strings: the URL pattern and the qualified class name of the front controller. The URL pattern in this example matches any request URL that ends with the *.scp* extension, either with or without query string. Because the syntax in property files follows regular expression rules, special characters such as '?' must be escaped with '\ backslash when used as literals. The word *preload* at the end signifies that the class is instantiated once at application startup and not upon every request. Multiple front controllers are separated by comma:

```
frontControllers=.*\.scp(\?.*)? myapp.frontcontrollers.AccessLogger preload,
.*\.scp(\?.*)? chronolog.frontcontroller.Authenticator preload
```

URL Rewriting

The SCP web framework provides a simple mechanism for URL rewriting similar to Apache HTTPD. This mechanism allows us to map application-dependent URLs to “fancy” URLs which are more readable, more relevant, and more search engine friendly. URL rewriting is likewise based on

javax Filters.

URL rewrite rules are optional. They are specified either in the configuration properties file using the *rewriteRules* parameter, or in a separate file named WEB-INF/rewriterules.txt. If they are specified in config.properties, the syntax is similar to that of front controller settings. If they are specified in rewriterules.txt, the following rules apply:

Each rewrite rule is a line composed of two strings separated by blanks or tabs: *from* and *to*. The *from* string is a regular expression that matches one or more URLs. The *to* string replaces the original URL. It may contain positional parameters (e.g. \$1, \$2, \$3) corresponding to groups in the *from* regular expression. The *from* expression is matched against the request URI beginning at the slash position after the root URL (=context path) and including the query string -if any-, up to the end of the query string. This means the root URL (=context path) is always ignored.

WEB-INF/rewriterules.txt example:

```
#
# Some sample rewrite rules:
#

# rewrites index.html in the products directory
# to products.scp in the root directory.
/products/index.html /products.scp

# rewrite requests for index.html in any directory
# to index.scp with optional query string
(.*)/index.html(\?.*)? $1/index.scp$2

# rewrite SEO-friendly product catalog URL into a request with query string
/catalog/products/(\d+)/(\w+) /catalog/product.scp?name=$2&article=$1
```

You can see from this example, that the URLs that users see are on the left side, while the application URLs are on the right side. For example, the “fancy” URL <http://domain.com/context/catalog/products/34716/actuator> is resolved to the application URL <http://domain.com/context/catalog/product.scp?name=actuator&article=34716>.

web.xml Deployment Descriptor

The deployment descriptor for the SCP web framework is fairly straightforward:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <display-name>Scala Pages</display-name>
  <description>
    Welcome to Scala Web Programming
  </description>

  <filter>
    <filter-name>ScpFilter</filter-name>
    <filter-class>scp.core.ScpFilter</filter-class>
  </filter>
  <filter-mapping>
```

```
<filter-name>ScpFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>

<servlet>
  <servlet-name>ScpServlet</servlet-name>
  <servlet-class>scp.core.ScpServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>ScpServlet</servlet-name>
  <url-pattern>*.scp</url-pattern>
</servlet-mapping>

<welcome-file-list>
  <welcome-file>index.scp</welcome-file>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

</web-app>
```

It contains definitions for ScpServlet and ScpFilter which together act as the dispatching mechanism of the SCP framework. If you make neither use of URL rewriting nor front controllers, you can dispense with ScpFilter; however ScpServlet is required in any case. The URL pattern for ScpServlet defines the extension used for dynamic pages and (implicitly) the extension of template files. Obviously, you can set this to anything you like.

Template Caching

SCP caches non-computed template files using a 2Q algorithm. This reduces IO access as frequently accessed templates are kept in memory. At this time, the output result is always recomputed for each request. Result caching may be introduced in later versions.