Package by feature, not layer

The first question in building an application is *"How do I divide it up into packages?"*. For *typical business applications*, there seems to be two ways of answering this question.

**Package By Feature**

*Package-by-feature* uses packages to reflect the feature set. It tries to place all items related to a single feature (and *only* that feature) into a single directory/package. This results in packages with high cohesion and high modularity, and with minimal coupling between packages. Items that work closely together are placed next to each other. They aren't spread out all over the application. It's also interesting to note that, *in some cases*, deleting a feature can reduce to a single operation - deleting a directory. (Deletion operations might be thought of as a good test for maximum modularity: an item has maximum modularity only if it can be deleted in a single operation.)

In package-by-feature, the package names correspond to important, high-level aspects of the problem domain. For example, a drug prescription application might have these packages:

* com.app.doctor
* com.app.drug
* com.app.patient
* com.app.presription
* com.app.report
* com.app.security
* com.app.webmaster
* com.app.util
* and so on...

Each package usually contains only the items related to that particular feature, and no other feature. For example, the com.app.doctor package might contain these items:

* DoctorAction.java - an action or controller object
* Doctor.java - a Model Object
* DoctorDAO.java - Data Access Object
* database items (SQL statements)
* user interface items (perhaps a JSP, in the case of a web app)

It's important to note that a package can contain not just Java code, but other files as well. Indeed, in order for package-by-feature to really work as desired, *all*items related to a given feature - from user interface, to Java code, to database items - must be placed in a single directory dedicated to that feature (and only that feature).

*In some cases*, a feature/package will not be used by any other feature in the application. If that's the case, it may be removed simply by deleting the directory. If it is indeed used by some other feature, then its removal will not be as simple as a single delete operation.

That is, the package-by-feature idea does not imply that one package can never use items belonging to other packages. Rather, package-by-feature **aggressively prefers package-private as the default scope**, and only increases the scope of an item to public only when needed.

**Package By Layer**

The competing *package-by-layer* style is different. In package-by-layer, the highest level packages reflect the various application "layers", instead of features, as in:

* com.app.action
* com.app.model
* com.app.dao
* com.app.util

Here, each feature has its implementation spread out over multiple directories, over what might be loosely called "implementation categories". Each directory contains items that usually aren't closely related to each other. This results in packages with **low cohesion and low modularity**, with high coupling between packages. As a result, editing a feature involves editing files across different directories. In addition, deleting a feature can almost never be performed in a single operation.

**Recommendation: Use Package By Feature**

*For typical business applications*, the package-by-feature style seems to be the superior of the two:

**Higher\_Modularity**   
As mentioned above, only package-by-feature has packages with high cohesion, high modularity, and low coupling between packages.

**Easier\_Code\_Navigation**   
Maintenance programmers need to do a lot less searching for items, since all items needed for a given task are usually in the same directory. Some tools that encourage package-by-layer use package naming conventions to ease the problem of tedious code navigation. However, package-by-feature *transcends the need for such conventions in the first place*, by greatly reducing the need to navigate between directories.

**Higher\_Level\_of\_Abstraction**   
Staying at a high level of abstraction is one of programming's guiding principles of lasting value. It makes it easier to think about a problem, and emphasizes fundamental services over implementation details. As a direct benefit of being at a high level of abstraction, the application becomes more self-documenting: the overall size of the application is communicated by the number of packages, and the basic features are communicated by the package names. The fundamental flaw with package-by-*layer* style, on the other hand, is that it puts implementation details ahead of high level abstractions - which is backwards.

**Separates\_Both\_Features\_*and\_*Layers**   
The package-by-feature style still honors the idea of separating layers, but that separation is implemented using separate *classes*. The package-by-*layer* style, on the other hand, implements that separation using *both* separate classes *and* separate packages, which doesn't seem necessary or desirable.

**Minimizes-Scope**   
Minimizing scope is another guiding principle of lasting value. Here, package-by-feature allows some classes to decrease their scope from public to package-private. This is a significant change, and will help to [minimize ripple effects](http://www.javapractices.com/topic/Topic123.cjp). The package-by-layer style, on the other hand, effectively abandons package-private scope, and forces you to implement nearly all items as public. This is a fundamental flaw, since it doesn't allow you to minimize ripple effects by keeping secrets.

**Better\_Growth\_Style**

In the package-by-feature style, the number of classes within each package remains limited to the items related to a specific feature. If a package becomes too large, it may be refactored in a natural way into two or more packages. The package-by-*layer* style, on the other hand, is monolithic. As an application grows in size, the number of packages remains roughly the same, while the number of classes in each package will increase without bound.

If you still need further convincing, consider the following.

**Directory Structure Is Fundamental To Your Code**

"As any designer will tell you, it is the first steps in a design process which count for most. The first few strokes, which create the form, carry within them the destiny of the rest." - *Christopher Alexander*

(Christopher Alexander is an architect. Without having worked as programmer, he has influenced many people who think a lot about programming. His early book*A Pattern Language* was the original inspiration for the Design Patterns movement. He has thought long and hard about how to build beautiful things, and these reflections seem to largely apply to software construction as well.)

In a CBC radio interview, Alexander recounted the following story (paraphrased here): "I was working with one of my students. He was having a very difficult time building something. He just didn't know how to proceed at all. So I sat with him, and I said this: Listen, start out by figuring out what the most important thing is. Get that straight first. Get that straight in your mind. Take your time. Don't be too hasty. Think about it for a while. When you feel that you have found it, when there is no doubt in your mind that it is *indeed* the most important thing, then go ahead and make that most important thing. When you have made that most important thing, ask yourself if you can make it more beautiful. Cut the bullshit, just get it straight in your head, if you can make it better or not. When that's done, and you feel you cannot make it any better, then find the *next* most important thing."

What are the first strokes in an application, which create its overall form? It is the directory structure. The directory structure is the very first thing encountered by a programmer when browsing source code. Everything flows from it. Everything depends on it. It is clearly one of the most important aspects of your source code.

Consider the different reactions of a programmer when encountering different directory structures. For the package-by-feature style, the thoughts of the application programmer might be like this:

* "I see. This lists all the top-level features of the app in one go. Nice."
* "Let's see. I wonder where this item is located....Oh, here it is. And everything else I am going to need is right here too, all in the same spot. Excellent."

For the package-by-*layer* style, however, the thoughts of the application programmer might be more like this:

* "These directories tell me nothing. How many features in this app? Beats me. It looks exactly the same as all the others. No difference at all. Great. Here we go again..."
* "Hmm. I wonder where this item is located....I guess its parts are all over the app, spread around in all these directories. Do I really have all the items I need? I guess we'll find out later."
* "I wonder if that naming convention is still being followed. If not, I will have to look it up in that other directory."
* "Wow, would you look at the size of this single directory...sheesh."

**Package-By-Layer in Other Domains is Ineffective**

By analogy, one can see that the package-by-layer style leads to poor results. For example, imagine a car. At the highest level, a car's 'implementation' is divided this way (package-by-feature) :

* safety
* engine
* steering
* fuel system
* and so on...

Now imagine a car whose 'implementation' under the hood is *first* divided up according to these lower level categories (package-by-layer) :

* electrical
* mechanical
* hydraulic

In the case of a transmission problem, for example, you might need to tinker around in these three compartments. This would mean moving from one part of the car to another completely different one. While in these various compartments, you could 'see' items having absolutely nothing to do with problem you are trying to solve. They would simply be in the way, always and everywhere distracting you from the real task at hand. Wouldn't it make more sense if there was a single place having exactly what you need, and nothing else?

As a second example, consider a large bureacracy divided up into various departments (package-by-feature):

* front office
* back office
* accounting
* personnel
* mail room

If a package-by-layer style was used, the primary division would be something like :

* executives
* managers
* employees

Now imagine the bureacracy being divided *physically* according to these three categories. Each manager is located, for example, with all the other managers, and not with the employees working for them. Would that be effective? No, it wouldn't.

So why should software be any different? It seems that package-by-layer is just a bad habit waiting to be broken.

The example applications that come with [WEB4J](http://www.javapractices.com/topic/TopicAction.do?Id=188) uses the package-by-feature style.

## Scope Of This Tutorial

This short tutorial shows you the basics of building a Java web application with WEB4J. It's particularly suitable for those with less experience with servlets and Java web apps. It's centered on the core task of building a simple web page interface to a simple relational database. It will give you a good feeling for what development with WEB4J will be like. If you have any suggestions for improving this tutorial, please [**send us a note**](mailto:support@web4j.com).

The tutorial has 2 parts:

* Part One covers the basics of servlets, and allows you to browse the code of a simple, minimal example application called Electricity Tracker.
* Part Two is about doing instead of reading. In Part 2 you deploy the Electricity Tracker application and add a new feature to it.

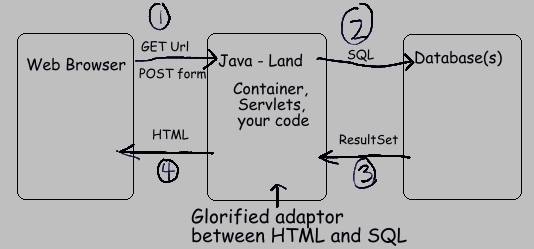
You will need to first download the source code for Electricity Tracker application (which includes web4j.jar):

**[Download Now](http://www.web4j.com/electricity/electricity.zip)**

This tutorial application is much is simpler than an average application. When you're done with the tutorial, you may want to [**download**](http://www.web4j.com/Download.jsp) the other example applications:

* Predictions - a medium-sized example app. It has security based on user, not role, like most public web sites.
* Fish & Chips Club - a larger example app. It has security based on role, like most intranet apps.

## The Basics Of Servlets



Web applications connect a web browser to database, and allow users to view and edit the data. The database is usually a relational database. [**Structured Query Language**](http://en.wikipedia.org/wiki/SQL) (SQL) is used to interact with a relational database. In addition, web applications are built on top of the simple request/response structure of [**HTTP**](http://www.w3.org/Protocols/).

### Directory Structure of a Web Application

A simple, static web site isn't connected to a database at all, and consists of ordinary web pages, written in HTML (hypertext). Those pages are usually arranged in a hierarchy, as in this example:

[document root directory]

index.html

tableOfContents.html

faq.html

essays/

baudelaire.html

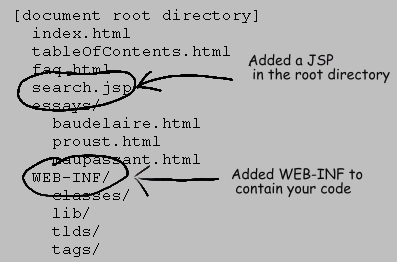
proust.html

maupassant.html

How can this be extended to allow for code? In Java, you create a **web application** by extending this basic directory structure in two ways:

1. allowing [**Java Server Pages**](http://www.oracle.com/technetwork/java/javaee/jsp/index.html) (JSPs) to be served in addition to HTML pages
2. adding a special directory named **WEB-INF**. The WEB-INF directory contains code that uses the [**Servlet API**](http://java.sun.com/products/servlet/) (directly or indirectly) to interact with the database. When the code is finished processing, it usually passes control to a JSP, which dynamically creates the response sent back to the browser.

If we changed the simple static web site shown above into a Java web application, then it would have this general form:



Although there are variations, the /WEB-INF/ directory typically has this structure:

/WEB-INF/ - contains web.xml (see below)  
/WEB-INF/classes/ - your Java classes, arranged in packages  
/WEB-INF/lib/ - Java libraries, in the form of .jar files  
/WEB-INF/tlds/ - .tld files, which configure tag libraries (used in JSPs)  
/WEB-INF/tags/ - .tag files, which are reusable JSP snippets (used in JSPs)

The **web.xml** file is important, and has the official name of **deployment descriptor**. It contains configuration information. It tells the container (see below) what it needs to know about your application, in a portable manner. It can also be used by your application, as a place to specify your own configuration items.

Items specified in web.xml include:

* names of your servlet classes
* how servlets map to URLs
* user roles and security constraints
* login forms
* error pages
* session timeout
* tag libraries
* and other items...

The HTML and JSP files under the root directory can still be served directly to a browser. All files under the /WEB-INF/ directory, however, have a special status. Since they contain your code, these files cannot be served directly to the browser. This protects you, since if the files were served directly, in the same manner as HTML files from a static web site, then that would pose a grave security risk to your application. Why? Because hackers could easily gain an enormous amount of information about your code.

Your web app, from the root directory on down, can be placed in a standard zip file, and given the extension **.war** (web app archive). Such **war files** are the most common way to deploy java web apps.

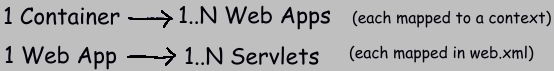
### Containers and URL Mapping

A web server that can run servlets is called a **servlet container** (often referred to simply as a container, for short). Apache [**Tomcat**](http://tomcat.apache.org/) is an example of a container. Application servers such as JBoss host Enterprise Java Beans (EJBs), but they also include servlet containers as well. The container handles all incoming HTTP requests, and decides which web application should handle each request.

Q: How does the container decide which application should handle each incoming HTTP request?   
A: It's actually quite simple. Each application is mapped to its own URL - or more precisely part of a URL, called a **context**. This is best explained with an example. Let's say you run a site called foodies.com on a Tomcat server. If that server explictly maps the context 'cheese' to web application X, then any URL of the form:

http://www.foodies.com/cheese/\*

will be passed by the container to that web application X. That is, the container will pass the incoming HTTP request to web app X for processing. The exact way a context is assigned to a web app depends on the container, but it's always a simple, straightforward mapping.



A Java web application is, in turn, made up of one or more servlets. Many web apps have a single servlet, but some use more than one. The Electricity Tracker has a single servlet. The Fish & Chips Club application has two - one for the main problem domain, and a second one to redirect directory requests to a designated home page.

Within each web app, the 'remaining' part of the URL (that is, the part that comes after the context) is used to map each request to a specific servlet. That mapping uses settings in your web.xml file. This mapping task is not quite as simple as the mapping of context-to-web-app, and can take various forms, according to your needs. One common style is to map a servlet to an extension such as '.do'. Let's take these example URLs to illustrate:

URL A: http://www.foodies.com/cheese/poll/viewresults.do

URL B: http://localhost:8080/electricity/main/home/SpendingAction.list

These URLs are split up by the container into three basic parts:

|  |  |  |
| --- | --- | --- |
|  | **URL A** | **URL B** |
| server | http://www.foodies.com | http://localhost:8080 |
| context | /cheese | /electricity |
| remainder | /poll/viewresults.do | /main/home/SpendingAction.list |

The context is used to decide which web app to call, and the "remainder" is used to map to a specific servlet within that web app. In many cases, all URLs under a given context will be handled by a single servlet.

### Scopes

The servlet API defines various **scopes** under which objects may be placed. Items are placed in these scopes as a name-value pair, where the name is a String, and the value is an arbitrary Java Object. Roughly speaking, this is just a way of passing data around. The most important instance of this is in your Actions. Your Action classes (see below) typically retrieve data from the database. To pass this data from the Action class to a JSP, where the data is rendered in HTML, the Action usually places the data in 'request scope' under a given key name. This allows the JSP to later reference the data, under the same name.

Placing an item in 'application scope' roughly means that a JSP or a class can get access to the item at all times. Placing items in 'request scope' and 'session scope' means that they can be accessed only in that request or that session, respectively.

### Java Server Pages and JSTL

The most common way of generating a textual response to a browser request is by using [**Java Server Pages**](http://www.oracle.com/technetwork/java/javaee/jsp/index.html) (JSPs). A JSP can serve any kind of text at all - plain text, HTML, XML,[**CSV**](http://en.wikipedia.org/wiki/Comma_separated_values) data, [**JSON**](http://www.json.org/) data, and so on. But the most common form of text to send back in a response is hypertext (HTML). The general idea is to mix standard, static hypertext together with dynamic items, to render the desired data. Here's an example of a JSP snippet which dynamically renders an HTML table. As you can see, the overall appearance of a JSP is not very different from ordinary hypertext.

<table class="report" title="Spending" align="center">

<caption>Electricity Spending</caption>

<tr>

<th title="Line Number">#</th>

<th title='Amount Paid'>Payment</th>

<th title='Kilowatt Hours'>KWH</th>

<th title='Estimated Reading'>Est</th>

<th title='Amount Per KWH'>Per KWH</th>

<th>Building Type</th>

<th>Comment</th>

</tr>

<c:forEach var="item" items="${itemsForListing}" varStatus="index">

<tr>

<td title="Line Number">${index.count}</td>

<td align="right">

<fmt:formatNumber value="${item.payment}" pattern='#,##0.00'/>

</td>

<td align='right'>

<fmt:formatNumber value="${item.kilowattHours}" pattern='#,##0.0'/>

</td>

<td align='left'>

<c:if test="${item.isEstimated}">

<span title='Estimated Reading'>E</span>

</c:if>

</td>

<td align='right'>

<fmt:formatNumber value="${item.paymentPerKWH}" pattern='#,##0.00'/>

</td>

<td align='left'>${item.facility}</td>

<td align="left">${item.comment}</td>

</tr>

</c:forEach>

</table>

Can you spot all the items in the above JSP snippet which are not standard HTML? The first row of the table contains column names, and is implemented with standard HTML. But the remaining rows are rendered dynamically, using a mixture of HTML and [**JSTL**](http://www.oracle.com/technetwork/java/index-jsp-135995.html) (JSP Standard Template Library). The <c:forEach>,<c:if>, and <fmt> tags are part of JSTL. So are all items appearing with the '${...}' syntax. For example, ${itemsForListing} refers to the collection of items being rendered in the table. This is an example of something mentioned earlier in the tutorial: a reference to a Java object (a List of Model Objects, in this case) which has been placed explicitly in "request scope" by an Action class, in order to be rendered in the JSP.

### Other Items to Note

People typically use a **framework** when building Java web apps, since the Servlet API is rather low-level. These frameworks exist in order to make the job of creating typical web apps easier. A framework is said to be **full-stack** if it helps you build all parts of your app (for example, WEB4J is a full-stack framework). Some tools focus on one area. For instance, Hibernate is not a full-stack framework since it's concerned solely with persisting your model to the database, and nothing else. Thus, some apps are created using more than one framework, such as Struts-plus-Hibernate.

For more information on Java web applications, and Java in general, here are some good references:

* [***Head First Servlets and JSPs***](http://www.amazon.com/exec/obidos/ASIN/0596516681/ref=nosim/javapractices-20) - Basham, Sierra, and Bates
* [***The Java Programming Language***](http://www.amazon.com/exec/obidos/ASIN/0321349806/ref=nosim/javapractices-20) - Arnold, Gosling, and Holmes
* [***Effective Java***](http://www.amazon.com/exec/obidos/ASIN/0321356683/ref=nosim/javapractices-20) - Bloch
* [***Head First Java***](http://www.amazon.com/exec/obidos/ASIN/0596009208/ref=nosim/javapractices-20) - Sierra and Bates

## 

## What's Included

The following items form part of the Electricity Tracker application:

* the Action, Model, DAO, .sql file, and JSP for a single feature
* a form with entry of a date, some numbers, some text, a drop-down select, and a boolean
* a list-and-edit form, with 5 SQL operations handled by one Action class
* display of success/fail messages to the end user
* displaying listings with alternating row styles, to increase legibility
* templated pages (header, footer, body, and so on)
* a user interface in a single language only (English)
* building Model Objects from request parameters
* form population with Model Object data
* logging to a text file
* a single database with two tables (one of which is a code table)
* 4 Java packages compatible with package-by-feature: domain, code tables, util, and config
* a practical example of using a simple code table extracted from the database
* an immutable Model Object that does its own validation
* a few settings in web.xml specific to web4j
* use of JSP/JSTL, and some web4j custom tags
* use of SafeText to protect from XSS hacks
* use of Decimal to represent money
* some .tag files are used to encapsulate markup

## What's Not Included

In order to keep things simple for this tutorial, the following items do **not** form part of the Electricity Tracker application:

* no security constraints or access control
* no login or logoff
* no CsrfFilter
* no user preferences
* no translation of user interface into multiple languages
* no pages used only by the webmaster
* no file uploads
* no multivalued parameters
* no unit tests
* no custom settings for various items - defaults used
* no database transactions
* no TroubleTicket emails sent when an error occurs

## List Of File Types

|  |  |  |
| --- | --- | --- |
| **File Type** | **Num Files** | **Comment** |
| .html | 4 | Plain HTML. |
| .jsp(f) | 3 | HTML plus JSP/JSTL, with some custom tags. |
| .tag | 5 | Reusable JSP snippets. |
| .java | 11 | Domain classes, plus some classes required by the framework. |
| .tld | 4 | Tag library files. You don't touch these. |
| .jar | 6 | Required Java libraries. Includes web4j.jar. |
| .xml | 2 | web.xml, and electricity.xml (the Tomcat context file). |
| .sql | 2 | SQL statements used by the app, and a script for creating the database. |
| .css | 1 | Cascading style sheet. |
| .ico | 1 | Image used by bookmarks. Referenced in the <head> of Template.jsp. |

## List Of Directories

|  |  |
| --- | --- |
| **Directory** | **Contains** |
| [ROOT] | Home page, error pages, stylesheet, icon. |
| main | A template JSP. |
| WEB-INF/ | The web.xml deployment descriptor. |
| WEB-INF/lib/ | Required jar files. |
| WEB-INF/mysql/ | Single database script for creating the database. |
| WEB-INF/tld/ | Contains only .tld files, which add custom tag libraries to your application's environment. Custom tags are used in JSPs. You don't typically touch these .tld files. |
| WEB-INF/tags/ | Contains only .tag files, which are encapsulated snippets of JSPs. |
| WEB-INF/classes/  hirondelle/web4j/config/ | Implementations of interfaces required by WEB4J. |
| WEB-INF/classes/  hirondelle/electricity/ | The bulk of the application's code. |

## 

## Tour Of All Items In The Implementation

This tour gives a summary of each and every file in the application - why it exists, and what it does. This part of the tutorial is a bit boring. It's really meant as a reference. **If you find it boring,**[**just skip ahead**](http://www.web4j.com/Tutorial.jsp#SystemRequirements)**. You can refer back to it later when needed.** If you are new to Java web apps, then you should read this section closely at some point.

**/Error.html, /ErrorAccessDenied.html, /ErrorBadRequest.html**  
Error pages. These exist to override the default error pages served by your container (Tomcat). The problem with such container default pages is that they are often a security risk, since they expose details that are useful to hackers. They're also ugly, and inappropriate for end users. It's a good idea for a web app to replace such default error pages.

These custom error pages are referenced in web.xml. In effect, the container is simply instructed to replace its own error pages with new versions specific to your web app.

**/index.html**  
The home page for the application. This file exists because requests such as

http://www.blah.com/

don't refer to a specific location or 'file' (it's actually often not a file.) Thus, the container needs instructions on what exactly to serve for such requests -- that is, what should be served by default. This item is configured in web.xml as a <welcome-file>, to be served when an incoming URL request specifies only a directory, and nothing else.

**/stylesheet.css**  
Standard Cascading Style Sheet. Exists in order to encapsulate styling decisions. Reduces repetition of styling information.

**/main/Template.jsp**  
Template for JSPs. Exists in order to define a shared layout in one place - header, body, footer, and so on.

The big idea here is that 'main' is a module that contains N features. (In the present instance, N=1, but the idea is that new features can be added to main just by following the same pattern used in main.home. This is in fact done in Part Two of this tutorial.) Those N features usually need to share the same look. This template defines that look. Each feature, in turn, provides its own version of the 'body' of Template.jsp.

(To clarify, 'module' is informally used here to refer to a collection of features that share something in common. For instance, a 'main' module might contain features needed by typical users, while a 'webmaster' module would contain features only accessible to an administrator.)

For comparison, in the Fish & Chips Club application there are several modules, each containing N features. Each module has its own look, so each module defines its own Template.jsp.

Again, this is one way of implementing templates. If, for example, you decide that all pages in the app need to have the same look, then you will need to slightly modify the given style (not hard - just don't use 'main').

You could explore ways to implement such templating using JSPs, but the style presented here is likely among the simplest.

**/WEB-INF/web.xml**  
The web.xml **deployment descriptor** is an important file. (See description above.) It contains your app's config. It instructs the container (Tomcat) on how to run your application. It contains a mixture of general items defined by the servlet API itself (used by the container), and items defined by WEB4J (used by the framework). In Electricity Tracker, the number of items in web.xml is relatively small, since, for simplicity, a number of WEB4J settings are left out. These WEB4J settings take default values when left out, but that is not true of web.xmlsettings in general.

For reference, the full set of WEB4J config items is listed [**here.**](http://www.web4j.com/UserGuide.jsp#ConfiguringWebXml)

You need to exercise some care when editing web.xml, since if you make a small typing mistake, the file will be rendered invalid, and your app will fail to start.

**/WEB-INF/JspHeader.jspf**  
Items that are included at the start of every JSP. This is a fragment of a JSP. It's not a complete JSP, but rather something that can be added to JSP. Hence, it has a different extension, '.jspf'. This item is referenced in web.xml, as part of jsp-config. That reference causes the container to place this JSP fragment at the start of every JSP. The fragment itself contains a character encoding, and some items which allow tag libraries to be referenced.

**/WEB-INF/TagHeader.jspf**  
This is very similar to JspHeader.jspf (see above), in that it exists to reduce repetition. This file is referenced in all .tag files. In this case, those references need to be added manually to each .tag file, since there's no mechanism in web.xml for doing so.

**/WEB-INF/lib/\*.jar**  
These .jar files contain libraries -- collections of Java classes. 'Jar' stands for 'java archive'. Jars are basically zip files containing compiled Java classes. At runtime, when the container is running your app and it encounters a class that has not yet been loaded, then it searches in several standard places for the implementation of the class. The servlet API specifies that one of those places is this /WEB-INF/lib/ directory. Your web apps will almost always have such a directory.

In the current case, the jars present in this directory are used by the web4j framework. But you can place other jars here as well, as needed. If, for example, your app needs file upload support, then you can add third party jars that support that task to this directory.

During development, you will need to point your IDE to these libraries, such that they are on your build path.

**/WEB-INF/mysql/CreateTables.SQL**  
Simple script for creating the MySQL database used by Electricity Tracker. This script is used only by the developer. It's not used at runtime, since at runtime the database has already been created.

It's mildly interesting to note why this item is placed here. It's not placed in the root directory, since that would be a grave security risk: the file could be served by the container, just like an HTML file in the root directory. That's a big no-no, since it would give hackers details about your app's underlying database. Since this file is below /WEB-INF/, however, which the servlet spec states is a 'protected area', the file can't be served automatically by the container.

Note as well the extension is '.SQL', not '.sql'. The reason for this is that WEB4J, upon startup, scans your application's source for all files that end in '.sql' (case-sensitive), and inspects their content for the SQL statements used at runtime by your app. Changing the extension to the upper case .SQL ensures that the file is not selected during that process.

**/WEB-INF/tags/displayMessages.tag**  
JSP snippet which renders success/fail messages to the end user. The messages are rendered with a <w:messages> tag.

<w:messages name="web4j\_key\_for\_errors">

<span class="error">placeholder</span><br>

</w:messages>

This tag uses its body as a simple template. It replaces the special 'placeholder' text with the actual message. This way, you can wrap your messages with arbitrary markup, to make them render as desired.

The messages come from your Actions. Calling addMessage or addError in your Action class will make the message visible to this tag. You can call these methods multiple times in your Action, and each message will be added to an internal list. Then, each message will be displayed here, one after the other, in the same order.

**/WEB-INF/tags/editLinks.tag**  
JSP snippet which creates links and buttons for editing and deleting items in a listing. Exists because it's common to have listings of items that can be edited, by either changing the item, or by deleting it. It's useful to encapsulate such items since they are often used throughout an app.

**/WEB-INF/tags/footer.tag**  
JSP snippet for the footer of each page. Uses an item named web4j\_key\_for\_app\_info. Each WEB4J app must provide an implementation of the ApplicationInfo interface. This tells WEB4J basic info about the app - name, version, and so on. Upon startup, the framework will use that implementation to create an object, and place it in 'application scope', under the name'web4j\_key\_for\_app\_info'.

**/WEB-INF/tags/head.tag**  
JSP snippet placed in the <head> tag of a JSP. Refers (somewhat obscurely) to param.TTitle. This is a request parameter that is defined by the templating mechanism used by theResponsePage class. The ResponsePage class is used by your app to tell the container which JSP is used to render the final result to the user. The ResponsePage can simply point to a single file, or, more commonly, it can point to a template JSP. The template defines the overall look for a given set of pages -- perhaps for the whole site, perhaps for a subset. It also uses two special request parameters,

* TTitle - the title of the page, placed in a <title> tag
* TTBody - the name of a second JSP which contains the main body of the page. The template JSP will 'look up' this second JSP, and insert it into its body. Hence, a set of pages share a common look or layout, but vary in the middle or body of the page.

The extra 'T' in 'TTitle' and 'TBody' is meant to avoid conflict with your own request parameter names. It also stands for 'template'.

**/WEB-INF/tags/setFormTarget.tag**  
In the list-and-edit style of user interaction used by Electricity Tracker, the same form is recycled for various operations. The form can be used either for an 'add' operation or a 'change' operation. In some cases, the form's action URL can depend on this add-versus-change logic.

This tag exists to encapsulate that logic. It calculates the form target, and places it in a variable named 'formTarget'. This variable is then used by the calling JSP. Thus, the calling JSPs don't need to repeatedly calculate the form's action URL.

Some forms will find this tag useful, some won't. For example, a search or query operation typically uses a single form action, so this tag wouldn't be useful in that case.

**/WEB-INF/tlds/\*.tld**  
These .tld files are tag library descriptors. They configure tag libraries. You typically just copy these files into this directory. You typically don't edit them. They can be used simply as documentation by the programmer, to find out what services are provided by the library.

If someday you create a tag library, you will need to create a corresponding .tld as well.

The c.tld, fmt.tld, and fn.tld are from the JSTL - the JSP Standard Tag Library, which is widely used. The web4j.tld is specific to web4j. It defines a number of custom tags you can use in your JSPs. Two common such tags are:

* <w:populate> - populate a form with data
* <w:alternatingRow> - present rows in a listing in alternating style, to improve legibility

**/WEB-INF/tomcat/electricity.xml**  
This file is for developers only. It's not used at runtime. It's a file specific to the Tomcat container. It exists because you need to define a context for a web application (as mentioned at the beginning of this tutorial). The method of doing this is specific to each container. See Part Two for how this file is used to deploy the Electricity Tracker application.

**/WEB-INF/classes/hirondelle/web4j/config/AppInfo.java**  
Required item. Exists because WEB4J requires that your app supply an implementation of the ApplicationInfo interface. A simple class, just carries basic data like name and version.

**/WEB-INF/classes/hirondelle/web4j/config/ConnectionSrc.java**  
Required item. WEB4J requires that your app supply an implementation of the ConnectionSource interface. When WEB4J's data layer interacts with your database, it needs a database connection in order to do this. There are many policies an application may take with respect to getting a database connection. This class exists to let you specify that policy in any way you want.

**/WEB-INF/classes/hirondelle/web4j/config/ConvertParamErrorImpl.java**  
Required item. WEB4J requires that your app supply an implementation of the ConvertParamError interface. WEB4J performs simple conversions from underlying request parameters (Strings) to various building block objects such as Date, BigDecimal, and so on. Since the incoming data is not completely under your control, these conversions can sometimes fail. This class exists because WEB4J needs to know how you want to report such conversion failures to the end user.

Note that these conversions do not form part of your Model Object validation, where you define the constraints being applied to user input. That's a different task, and is handled in another way (see Spending.java).

**/WEB-INF/classes/hirondelle/web4j/config/DateConverterImpl.java**  
Required item. WEB4J requires that your app supply an implementation of the DateConverter interface. Deals with parsing and formatting dates. It's natural for an app to define a specific format for dates, such that the end user can repeatedly use the same format when entering dates. Since dates can be treated in so many different ways, the framework lets you decide exactly how you would like to do this.

**/WEB-INF/classes/hirondelle/web4j/config/Startup.java**  
Required item. WEB4J requires that your app supply an implementation of the StartupTasks interface. Exists because when your app starts up, there are often one-time initialization operations that need to happen. A common example is placing code tables in memory.

**/WEB-INF/classes/hirondelle/web4j/config/TranslatorImpl.java**  
Required item. WEB4J requires that your app supply an implementation of the Translator interface. Exists in order to implement multilingual applications. The current implementation is a simple do-nothing class, since this application has only one language.

If, in the future, you decide to make your app multilingual, then:

* the TranslatorImpl class would be reimplemented.
* in most cases, your Java classes would remain the same.
* several custom WEB4J tags, created specifically for translation, would need to be used in most JSPs. Those custom tags make use of this configured Translator.

**/WEB-INF/classes/hirondelle/electricity/codes/code\_table.sql**  
SQL SELECT statement for fetching code tables. Here, there's only one code table, called FACILITY. It refers to residential or commercial properties. It exists simply because almost all applications use code tables, and even a minimal application should have one. There are 2 tables in the database -- a code table and regular domain table.

This code table is fetched only once, upon startup, and then placed in memory. It's also placed in 'application scope' under a specific key. This makes it easy to reference in JSPs.

There are other variations on code tables that are possible.

**/WEB-INF/classes/hirondelle/electricity/codes/CodeTable.java**  
Exists in order to enumerate all the code tables in the application. Here, there's only one such table. But, you can extend this scheme to many code tables in a natural way, just by following the pattern.

Uses the Id and Code classes. Id is a generic sort of identifier class. Code is intended specifically for code tables, but you aren't required to use it to implement your own code tables.

Note that this class is an enumeration class. As well, it has methods that allow the caller to translate, as it were, an Id into a Code. This is useful in Model Objects, where the underlying request parameter is an Id, but is modeled as a Code. It's advantageous to model it as a Code, since the text description is immediately available. The text description is presented to the user. It's also useful for logging and debugging.

**/WEB-INF/classes/hirondelle/electricity/codes/CodeTableUtil.java**  
Exists in order to pull in code tables upon startup. Places code tables in memory, in a form which makes lookups convenient. Again, here there is only one such code table, but this class could be easily extended to more, just by following the pattern.

**/WEB-INF/classes/hirondelle/electricity/main/home/Spending.java**  
Model Object. Exists in order to encapsulate and validate the data being stored in the Spending table.

This is an example of an [**immutable object**](http://www.javapractices.com/topic/TopicAction.do?Id=29) -- an object whose data never changes once it's constructed. The constructor is the most important method. Note that it must throw aModelConstructorException. This is how your Model Objects communicate problems to the caller.

The toString, equals, and hashCode methods are defined in the Object class. It's always prudent to override these methods. They ensure desirable behavior for logging and Collections.

The getSignificantFields method exists since equals and hashCode need to [**refer to the same items**](http://www.javapractices.com/topic/TopicAction.do?Id=17).

**/WEB-INF/classes/hirondelle/electricity/main/home/SpendingAction.java**  
The Action class for this feature. The Action is the public face of each feature. It calls all of the other elements of the feature, directly or indirectly.

Your Actions will usually subclass a "template" abstract base class, defined by the framework. Several such template classes have been defined by WEB4J, but you can certainly define your own, if desired.

The SqlId declarations define what underlying SQL operations are performed by the Action. They exist because the text .sql files are not compiled as Java files. Thus, a bridge of some sort is required between .sql-land and Java-land. These SqlId declarations form that bridge. Once declared, they also allow you to later refer to SQL statements using Java objects instead of repeating magic Strings all over the place.

Upon startup, WEB4J scans your code for all such SqlId fields. They are matched one to one to corresponding items in your .sql files. If any mismatch is detected, then your app will not run. This protects you from simple typographic errors associated with magic Strings. It also protects your app from including items that are no longer used.

If you wish, these SqlId fields could be declared in Data Access Object (DAO) instead, but that would often mean that the DAO would need to change from package-private to public.

The RequestParameter declarations define the white list of accepted request parameters. If this Action encounters any unknown request parameter, an error will result. This is an important security measure.

RequestParameter objects define hard validations (length check, regex check). These are different from the soft validations performed in your Model Objects. See the [**User Guide**](http://www.web4j.com/UserGuide.jsp#ApplicationSecurity) for more information regarding this important distinction.

The addError and addMessage methods are called to pass success/fail messages to the end user.

The implementations of the various attemptXXX methods can vary in details. According to context, you may need to show special concern for duplicate records, missing records, the number of edited items, and so on.

**/WEB-INF/classes/hirondelle/electricity/main/home/SpendingDAO.java**  
Data Access Objects (DAOs) exist in order to encapsulate how you persist your Model Objects.

Each operation has its own method in this class, and its own underlying SQL statement. The SQL statement is referenced using SqlId objects. These SqlId objects in turn point to entries in your .sql files.

In this case, every method uses the Db utility class. When passing data to the methods of the Db class, the order in which data is passed corresponds directly to the order of appearance of the '?' placeholders in the underlying SQL statement.

**/WEB-INF/classes/hirondelle/electricity/main/home/statements.sql**  
Exists to define the SQL statements used by this feature. Usually, each feature will have its own statements.sql. Each SQL statement is placed in a named block, similar to a block in Java, as in

DELETE\_SPENDING {

DELETE FROM Spending WHERE Id=?

}

To act as a bridge between your .sql files and Java, a SqlId object corresponding to each such named block is declared somewhere in your code, as in

public static final SqlId DELETE\_SPENDING = new SqlId("DELETE\_SPENDING");

This declaration is usually declared in the feature's Action. If you place the declaration in your DAO, then the DAO will need to be made public (not package-private). The String passed to the*SqlId*constructor must match the name of a block in an .sql file.

**/WEB-INF/classes/hirondelle/electricity/main/home/view.jsp**  
Exists to provide the user interface for this feature. Consists of a fragment of Java Server Page (JSP). This fragment is passed to /main/Template.jsp as the 'body' section of the template. Usually, each feature will have its own view.jsp.

Most features will follow this style. When needed, it's also possible to refer to a full JSP directly, without any templating.

System Requirements

(This begins Part 2 of the tutorial, where you are doing instead of just reading.)

The System Requirements for this tutorial are:

* JDK 1.6 (or better)
* [**Tomcat**](http://tomcat.apache.org/whichversion.html) servlet container (Tomcat 6 or better)
* [**MySQL**](http://dev.mysql.com/downloads/) 5
* MySQL's Java database driver from [**mysql.com**](http://dev.mysql.com/downloads/connector/j/)
* The [**electricity.zip**](http://www.web4j.com/electricity/electricity.zip) file from webj.com.

The electricity.zip file includes web4j.jar. It also includes compiled .class files, which means that, initially, you may run the app without needing to compile. Any additions or changes you make to the code will, of course, will require a recompile.

**It's highly recommended that you do *not* use an IDE for this tutorial.** If you use an IDE, and a problem occurs, then it won't be clear if the problem is related to the IDE or to WEB4J. To aid with compilation, the required javac command line is provided below for your use.

It's strongly recommended that you use MySQL for this tutorial. If you wish to use a database other than MySQL, you'll need to do a little extra work on your own:

* change CreateTables.SQL to work with your database (this is usually easy).
* change a connection string setting in electricity.xml to point to your database.
* settings in web.xml may need to change. In particular, you may need to set HasAutoGeneratedKeys to false.

## Deploying To MySQL and Tomcat

Now, let's do a bit of work. We will deploy the Electricity Tracker to Tomcat and then run it. Next, we will do some coding, and add a simple feature.

In general, deployment is something whose details depends on the servlet container you are using.

There are different ways to deploy web applications to Tomcat. Here, we're going to deploy the app to Tomcat in such a way that any changes we make will be automatically detected by the container. This allows the container to detect changes to JSPs, without needing to restart the app. It will also speed up development since it will not actually be necessary to create a .war file.

(Variations are possible. In particular, you can change the extension of electricity.zip file to .war, and deploy it as any war file. However, you should attempt such variations only if you are experienced with Tomcat. It's highly recommended to follow the style used below.)

**1. Unzip the file.**  
Unzip electricity.zip to any location. Since you will be developing, it's a good idea to place it in an area you typically develop in. For example:

C:\myname\projects\electricity\

Just to be clear, this directory will contain:

/main

/WEB-INF

Error.html

...and so on

In the rest of this document, this directory is referred to as ELEC\_HOME.

**2. Create the database.**  
Run the provided database script, using your local mysql client. (Your user name will need to have wide privileges.) It creates the database, and its 2 tables. Here's an example of running the script. You will need to change the highlighted parts, to point to your own mysql.exe, and to use your own user name and password. Note that there's no space between the-p and the password.

[ELEC\_HOME]\WEB-INF\mysql>C:\mysql\bin\mysql --local-infile=1 -u myname -pmypass < CreateTables.SQL

**3. Put the database driver where Tomcat can see it.**  
We need to give Tomcat access to the database driver classes. You do that by placing the database driver (a single jar file) in Tomcat's lib directory, as in:

[TOMCAT\_HOME]\lib\mysql-connector-java-5.1.14-bin.jar

Please use the official JDBC driver, available from [**mysql.com**](http://dev.mysql.com/downloads/connector/j/).

**4. Set the logging directory in web.xml.**  
Open the file [ELEC\_HOME]/WEB-INF/web.xml. Edit the value of the LoggingDirectory. Set it to some suitable directory, if you don't like the default.

**5. Create the Tomcat context.**  
The file:

[ELEC\_HOME]\WEB-INF\tomcat\electricity.xml

is a Tomcat context file, which tells Tomcat what it needs to do when deploying the app:

* the location of the web app's implementation on the local file system.
* how to create a connection pool.
* what URL context to apply to the app (the /electricity/ context).

The URL context is deduced simply from the file name. When Tomcat sees an incoming HTTP request URL starting with http://localhost:8080/electricity/, it will forward it to our app, because the name of the context file is electricity.xml.

**You must edit electricity.xml for the following items:**

* the docBase attribute must point to ELEC\_HOME, the directory where you unzipped electricity.zip
* a valid MySQL user name and password must be set for the connection pool
* if your MySQL server is not running on the default port 3306, then change 3306 to the correct value

Here's an example of a customised version of electricity.xml. Compare it with your version, to see where you need to make changes specific to your own environment:

<Context docBase="C:\Temp4\electricity" reloadable="true">

<Resource

name="jdbc/electricity"

auth="Container"

type="javax.sql.DataSource"

username="johanley"

password="\*\*\*\*\*\*\*\*"

driverClassName="com.mysql.jdbc.Driver"

url="jdbc:mysql://localhost:3306/electricity?useServerPrepStmts=false"

maxActive="10"

maxIdle="5"

/>

</Context>

An important point is that C:\Temp4\electricity is, in this case, ELEC\_HOME. It contains the unzipped download:

C:\TEMP4\electricity>dir /B

Error.html

ErrorAccessDenied.html

ErrorBadRequest.html

favicon.ico

index.html

LICENSE\_BSD.txt

main

stylesheet.css

WEB-INF

If you have chosen to use postgres instead of MySQL, please see this [**remark**](http://www.web4j.com/UserGuide.jsp#Postgres) about passing an extra parameter to the connection URL.

**6. Deploy the context to Tomcat.**  
After you have made the above edits, copy the electricity.xml file to this location (if the directory doesn't exist, just create it):

[TOMCAT\_HOME]\conf\Catalina\localhost\

**7. Start the application.**  
If Tomcat is already running, then Tomcat will automatically detect the new context file after a few seconds, and your app is now deployed to Tomcat. If Tomcat is not running, then start it now. Navigate to the home page of the app. The typical URL will be (using the default Tomcat port of 8080):

[**http://localhost:8080/electricity/**](http://localhost:8080/electricity/)

Check your log file as well. There should be a timestamped file in the logging directory you specified in web.xml, in a previous step. This logging entry indicates a successful startup:

CONFIG: \*\*\* SUCCESS : STARTUP COMPLETED SUCCESSFULLY. \*\*\* ...

Please review the log file, and scan it for any problems. Most problems will be related to the database connection pool and the database jar, so please check those items carefully.

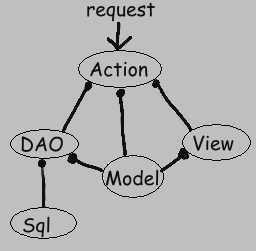
## Adding a New Feature - Overview

We are now going to add a new feature to the application. This feature will be simpler than most -- a simple report, showing the records in the Spending table that have the Amount as 100.00 or more.

This report will be implemented using 4 items, all placed in the same directory:

* a statements.sql file containing a single SELECT statement with no parameters
* a view.jsp for rendering the result
* a Data Access Object (DAO) to translate the SELECT result set into a List of Model Objects (Spending.java, already defined). As you will see, the DAO class is small. It could be refactored into the Action, if desired.
* an Action to bring it all together, using the other items in the feature's implementation

Here is how the parts work together (the notation means that the DAO uses the Model, for example):



**Create a New Package/Directory.**  
Create a new package/directory:

[ELEC\_HOME]\WEB-INF\classes\hirondelle\electricity\main\report\

All items related directly to this feature, and only this feature, will be placed in this single directory. This is an example of package-by-feature.

**Create the .sql File.**  
Create an ordinary text file named statements.sql, located in our feature's directory. It will contain the following:

LIST\_HIGH\_SPENDING {

SELECT Id, DatePaid, Amount, KWH, IsEstimated, FacilityFK, Comment

FROM Spending

WHERE AMOUNT>=100

}

This is a plain, ordinary SQL statement. The only thing unusual about it is that it's placed in a block named LIST\_HIGH\_SPENDING. This block name forms an identifier for that SQL statement. (Typically, there's more than one such block per file.)

**Create the View.**  
Create a file named view.jsp and place it into our feature's directory.

Copy the following into this file:

<P>

<table class="report" title="Spending" align="center">

<caption>

High Spending Report -- Num Lines: ${fn:length(itemsForListing)}

</caption>

<tr>

<th title="Line Number">#</th>

<th>Date Paid</th>

<th>Amount</th>

<th>KWH</th>

<th>Per KWH</th>

<th>Estimated</th>

<th>Building Type</th>

<th>Comment</th>

</tr>

<w:alternatingRow>

<c:forEach var="item" items="${itemsForListing}" varStatus="index">

<tr class="row\_highlight">

<td title="Line Number">${index.count}</td>

<td align='right'>

<c:set value="${item.datePaid}" var="paidOn"/>

<w:showDateTime name='paidOn' pattern='MMM DD YYYY'/>

</td>

<td align="right">

<fmt:formatNumber value="${item.payment}" pattern='#,##0.00'/>

</td>

<td align='right'>

<fmt:formatNumber value="${item.kilowattHours}" pattern='#,##0.0'/>

</td>

<td align='right'>

<fmt:formatNumber value="${item.paymentPerKWH}" pattern='#,##0.00'/>

</td>

<td align='left'>

<c:if test="${item.isEstimated}">E</c:if>

</td>

<td align='left'>${item.facility}</td>

<td align="left">${item.comment}</td>

</tr>

</c:forEach>

</w:alternatingRow>

</table>

*Save this file using the UTF-8 encoding.* (This will allow your app to support characters from just about any language.) Let's break this down a little bit.

<c:forEach>, <c:set>, <c:if>, <fmt:formatNumber>, fn:length   
These items are part of the JSTL - the JSP Standard Tag Library. JSTL is widely used in JSPs.

<w:alternatingRow>   
Custom tag defined by WEB4J for alternating the display of table rows. The idea here is that rows which alternate in appearance are more legible.

<w:showDateTime>   
Custom tag defined by WEB4J for displaying dates in the desired format.

${item.comment}, and so on   
This is the syntax used by the Expression Language defined in JSP 2.0. It's a compact way of referring to objects you have placed in scope.

**Create the DAO.**  
Create a Java source file named ReportDAO.java, in our feature's directory. Copy the following into it:

package hirondelle.electricity.main.report;

import java.util.\*;

import hirondelle.electricity.main.home.Spending;

import hirondelle.web4j.database.DAOException;

import hirondelle.web4j.database.Db;

import static hirondelle.electricity.main.report.ReportAction.LIST\_HIGH\_SPENDING;

final class ReportDAO {

List<Spending> list() throws DAOException {

return Db.list(Spending.class, LIST\_HIGH\_SPENDING);

}

}

This class will not yet compile, since it depends on an item in ReportAction, and that class hasn't been created yet. Its single method returns what we are looking to display in a JSP - aList of Spending Model Objects.

This class is package-private. It's not used by any other feature/package, so there is no need to make it a public class. It's always best to minimize scope whenever possible.

Let's take a look at the Db.[**list**](http://www.web4j.com/web4j/javadoc/hirondelle/web4j/database/Db.html#list(java.lang.Class,%20hirondelle.web4j.database.SqlId,%20java.lang.Object...))() method. The first parameter is a class literal. It defines the class of the object to be returned in the List. The next param, LIST\_HIGH\_SPENDING, identifies the SQL statement to be used to create these objects.

There's actually one other parameter to this method. It's a sequence parameter (or vararg). It stands for 0 or more pieces of data to be passed to the SQL statement. In this case, the SQL statement has no parameters, so no more arguments appear in this particular call to Db.list().

**There's an important ordering convention being used here behind the scenes.** Compare the columns returned by the underlying SQL statement with the Model Object constructor:

SELECT

Id, DatePaid, Amount, KWH, IsEstimated, FacilityFK, Comment

FROM

Spending

WHERE

Amount>=100

public Spending(

Id aId, Date aDatePaid, BigDecimal aAmount, Integer aKiloWattHours,

Boolean aIsEstimated, Id aFacility, SafeText aComment

) throws ModelCtorException {

...

}

Q: Can you spot the ordering convention?   
A: The columns returned by the SELECT match one to one with the constructor arguments. It's this ordering convention which allows your DAO to be so compact, since no intermediate mapping is needed. It's important that you understand this point. If the order of columns returned by a SELECT doesn't match the items passed to the constructor, then errors will result.

(It's possible to create a report without having a Model Object. That is, if a Model Object doesn't already exist, you don't have to create one just for a report. See the [**Report**](http://www.web4j.com/web4j/javadoc/hirondelle/web4j/database/Report.html) class for more information.)

**Create the Action.**  
Create a Java source file named ReportAction.java, in our feature's directory. Copy the following into it:

package hirondelle.electricity.main.report;

import hirondelle.web4j.model.AppException;

import hirondelle.web4j.action.ActionImpl;

import hirondelle.web4j.request.RequestParser;

import hirondelle.web4j.action.ResponsePage;

import hirondelle.web4j.database.SqlId;

public final class ReportAction extends ActionImpl {

public static final SqlId LIST\_HIGH\_SPENDING = new SqlId("LIST\_HIGH\_SPENDING");

public ReportAction(RequestParser aRequestParser){

super(FORWARD, aRequestParser);

}

@Override public ResponsePage execute() throws AppException {

ReportDAO dao = new ReportDAO();

addToRequest(ITEMS\_FOR\_LISTING, dao.list());

return getResponsePage();

}

// PRIVATE

private static final ResponsePage FORWARD = new ResponsePage(

"Example Report", "view.jsp", ReportAction.class

);

}

Action classes are always public, since they need to be visible to framework classes. ActionImpl is provided as a convenient base class for all Actions. It has a number of utility methods commonly required when implementing Actions. Your Action can extend it directly. However, it's often easier to extend other base classes provided by the framework (ActionTemplateXXX), since they may more closely match your needs. Which one you choose depends mostly on what set of operations the user will accomplish through the Action.

The SqlId field is declared here, but, if you wish, you can just as easily declare it in the DAO. The only reason for declaring it here is to keep the DAO package-private. Since SqlId fields need to be visible to the framework, they must appear in a public class.

Your Actions will typically take a RequestParser in their constructor. That object is constructed for you by the framework. It's used by ActionImpl to allow access to the underlying request.

The FORWARD object is a ResponsePage object. In this case, the templating mechanism is invoked. This report is in the main module, and shares the same Template.jsp as in the other feature, contained in hirondelle.electricity.main.home. The Template.jsp defines the layout, while view.jsp provides the main body of the page.

The single execute method uses your DAO to fetch the desired List of Spending objects. The List is then placed in "request scope" under the key named ITEMS\_FOR\_LISTING. Placing it in request scope means that it will be accessible in a JSP under the given key name.

The key name ITEMS\_FOR\_LISTING is defined in ActionImpl for your use. It's just a String, with the generic value "itemsForListing". In the JSP, it is referenced using JSP Expression Language as${itemsForListing}. If you wish, you can place Model Objects in request scope using any key name you like. For instance, in this case you might have used the key name "largeSpendingList" instead. In the JSP, that would be referenced using Expression Language as ${largeSpendingList}.

**Compile Your Classes**  
Next, make sure your application compiles successfully. Navigate to the directory

[ELEC\_HOME]/WEB-INF/classes/

and follow the example javac command contained in

[ELEC\_HOME]/WEB-INF/classes/javac\_compile\_example.txt

The command in this file must be run from the directory in which it resides.

You will likely need to edit this example javac command, to suit your environment:

* point to your JDK 5's javac.exe
* point to your Tomcat's servlet-api.jar

It's important to note that the generated .class files are placed in their default location -- in the same directory as the corresponding .java file. The reason is the standard directory structure of Java web applications mentioned above. Your compiled classes need to be located under /WEB-INF/classes, or the servlet container will not be able to find your code. Thus, it's not correct to place your compiled classes in a bin directory, for instance.

**Add a Link.**  
All items are now in place. The only thing that remains is to create a link pointing to the new feature's URL. Open

[ELEC\_HOME]/main/Template.jsp

and add a new link, just below the others:

<c:url value="/main/report/ReportAction.list" var="reportURL"/>

<A href='${reportURL}' title='Report on high spending'>Report</a>

It's possible to do this without using <c:url>. However, using <c:url> to emit links is highly recommended. It protects you from problems with URL rewriting, and allows you to ignore what context your app is deployed under.

Next, save Template.jsp, and restart the application. The restart is needed since WEB4J maps URLs to Action classes only upon startup. When you add a new Action, WEB4J needs a refresh to pull in the new mapping.

Navigate to the main page :

[**http://localhost:8080/electricity/main/home/SpendingAction.list**](http://localhost:8080/electricity/main/home/SpendingAction.list)

You should see a new menu item called Report. Click on Report, and you should see the report listing. For reference, the direct link is:

[**http://localhost:8080/electricity/main/report/ReportAction.list**](http://localhost:8080/electricity/main/report/ReportAction.list)

You should see the number of rows in the report (0 or more), and a listing of all Spending records whose amount is 100.00 or more. **If you see the report, then congratulations, you've just implemented your first feature using WEB4J. Cha-ching!**

To test further, click on the the Edit menu item and simply add some records having sufficiently large amounts. When you view the report again, any items with large amounts will appear in the report.

## Where To Go From Here

To learn more about WEB4J, please start with the [**documentation**](http://www.web4j.com/Support.jsp) page. Items are listed there in the order that most people will need to learn WEB4J. It starts with general ideas, and then gets increasingly more detailed.

The authoritative documentation should be taken as the [**User Guide**](http://www.web4j.com/UserGuide.jsp), and the WEB4J [**javadoc**](http://www.web4j.com/web4j/javadoc/index.html).

There are two web4j example applications to choose from, one called *Predictions*, and one called *Fish and Chips Club*. When starting with web4j, the *Predictions* application is recommended, because:

* it's smaller
* it uses a single language in the user interface, so you're less likely to encounter encoding issues
* it's more suitable for learning

The example applications aren't small. **Those less experienced with Servlets will greatly benefit from first going through the**[**tutorial**](http://www.web4j.com/Tutorial.jsp)**.**

Here's a **high level comparison** of the two example applications:

|  |  |  |
| --- | --- | --- |
|  | ***Predictions*** | ***Fish and Chips Club*** |
| Code Size | 42 classes | 99 classes |
| UI Language | English | English and French |
| Number of Databases | 1 | 3 |
| Encoding | UTF-8 | UTF-8 |
| Database | MySQL 5 | MySQL 5 |
| Access Control | Per User (web style) | Per Role (intranet style) |
| [**reCaptcha**](http://www.recaptcha.net/) Account | Required | Not Required |
| Javadoc | [**link**](http://www.web4j.com/predict/javadoc/index.html) | [**link**](http://www.web4j.com/fish/javadoc/index.html) |
| License | BSD Open Source | BSD Open Source |
| Download Link | **[Download Predictions Example App](http://www.web4j.com/predict/predictions.zip)** | **[Download Fish and Chips Club Example App](http://www.web4j.com/fish/fish.zip)** |

The web4j.jar binary is included in the example applications (and in the [**tutorial**](http://www.web4j.com/Tutorial.jsp)) at this location:

/WEB-INF/lib/web4j.jar

The 'look' of the example apps is simple - too simple for most real applications. It's important to note that WEB4J *is a Java API*, and it has little to say about the appearance of your web app. That is, you control the appearance of your apps using Cascading Style Sheets - not WEB4J.

To install the example applications, please follow closely the guidelines of the [**Getting Started Guide**](http://www.web4j.com/GettingStartedGuide.jsp).

In general, WEB4J applications can interact with any relational database. The example applications, and their documentation, use MySQL. A script is provided for the creation and population of their databases. For the example apps, you're encouraged to use MySQL if possible. If that's not possible, then you will need to do more work to get them running, by reviewing and editing all items related to the database:

* the provided script for creating and populating the database tables
* the datasource settings in the context configuration file (predictions.xml, fish.xml)
* the numerous settings in web.xml related to the database

If you decide to use a postgres database, see this [**remark**](http://www.web4j.com/UserGuide.jsp#Postgres) on adding a parameter to the connection URL.

**System requirements** are:

* JDK 1.6+
* Servlets 2.4+ (Tomcat 6+, for example)
* JSP 2.0+
* MySQL 5

The javadoc (see table above) includes convenient links to source code, JSPs, and SQL statements. The source code is syntax-highlighted to make it easier to read.

The example applications are provided as both a starting point and as a guide. It's recommended, though not required, that your applications be created by starting with an example app, and then changing it gradually. If any items are undesired, then they are simply removed by deleting directories (and any links from menus). This is only possible because of the [**package-by-feature**](http://www.javapractices.com/topic/TopicAction.do?Id=205) style recommended by WEB4J. (See the [**User Guide**](http://www.web4j.com/UserGuide.jsp#BuildingNewAppsFromTheExampleApp) for more information.)

This approach is consistent with the following:

* destroying things is easiest of all
* next, changing something that already exists is *usually* easier than starting with nothing
* finally, starting with nothing usually takes the most time

In any case, the classes in the example application are always effective guides.

The *Fish & Chips Club* serves as an informal "reference implementation", and exercises about 95% of the WEB4J API. It includes several modules:

* a main business domain
* webmaster tools - diagnostics, logging settings, performance stats
* user preferences - changing language, changing passwords
* access control - adding users, changing roles, resetting passwords
* translation - adding and editing translations for user interface text

**Other Downloads**

The following downloads are also available:

* *Predictions* [**Javadoc.zip**](http://www.web4j.com/predict/javadoc.zip)
* *Fish & Chips Club* [**Javadoc.zip**](http://www.web4j.com/fish/javadoc.zip)
* WEB4J [**Javadoc.zip**](http://www.web4j.com/web4j/javadoc.zip)
* WEB4J [**Dev Tools**](http://www.web4j.com/DevTools.jsp)
* WEB4J [**Jar File**](http://www.web4j.com/web4j/web4j.jar) (already included in the tutorial and example applications)
* WEB4J [**Source Code**](http://www.web4j.com/web4j/web4jsource.zip). The source files use UTF8 encoding. If you compile the source, you may need to [**specify UTF8**](http://java.sun.com/javase/6/docs/technotes/tools/windows/javac.html) as the file encoding.

For those stuck on JDK 1.5, the last version of web4j to be created using JDK 1.5 was 4.9.0:

* WEB4J 4.9.0 [**Source Code**](http://www.web4j.com/web4j/web4jsource-4.9.0.zip)
* WEB4J 4.9.0 [**Jar File**](http://www.web4j.com/web4j/web4j-4.9.0.jar)

All items are released under a BSD License.