

Spring 3.0 Web Applications

In this article, we are going to walk you through building a Spring 3.0 web application, including the usual suspects Hibernate, JSPs and automated testing.

The aim of this article is to go outline the best practices and show some interesting – dare we say *cool* – features you can use in your Spring 3.0 web applications. The code in this article works with Spring 3.0 M2; we tried to restrict the Spring 3.0 features to the ones we suspect may make it all the way to the final version of Spring 3.0.

Without further delay, let's take a look at what we are going to cover in the article:

- Maven-style directory structure
- Bundles (JARs, actually, but humour us)
- Annotation and XML configuration
- SPEL
- Spring MVC
- Automated build

What are we building?

Before we dive into the implementation of the application, let's take a look at what we're actually building. The product will be a web application that will demonstrate how to use some of the new (and older) features in Spring 3. We will show a listing page (including paging), a form page (including validation). Figure 1 shows the application in its full glory.

Figure 1. The application's user interface

Yummies

Name	Complexity	TTC	Cost
Mackerel with lettuce	Very easy	0:01	£ 2
Pasta with tuna	Very easy	0:15	£ 3
Butternut squash risotto	Easy	0:30	£ 4
Scallops with pea puree	Moderate	0:25	£ 10
Czech dumpling	Ridiculously hard	1:20	£ 5

1 2 3 4 5 6 ... 21 22 Next

Name * Required Field

Complexity 

Time to cook

Cost

Even though the user interface should be self-explanatory, let's review its main components. The listing page (under the Yummies header) shows a pageable list of all items in the food database. It should be possible to filter and order the results on all columns. The users should be able to create new, edit and delete any entry in the database.

Naturally, we will try to test every line of code using the most efficient testing approaches.

Directory structure

Even though you may not think it, the directory structure often dictates the complexity of the build process and ease of restructuring the application. Good directory structure makes it easy to:

- Identify where each source code file is,
- Automate build,
- Enforce architectural constraints,

With this in mind, let's consider our application. We will need some domain classes, repository (the cool word for data access) code, services that define and implement what we can do with the food items and a web application.

If you just put all packages in one directory, it is not the best approach, because in this one directory, you cannot easily enforce the architectural constraints. An example of such constraint is that it should be impossible to directly use the code in repository tier from the code in the web tier.

Another option is to create a directory for each tier. Even though this solves the problem from the previous paragraph, it too is not the best way of structuring your code. We should be ready to embrace change, be agile in our development process. We do not like to embark on long and complex projects with a "big bang" delivery. Instead, we like to deliver our applications in small, well-defined steps. Therefore, splitting the directory structure into tiers is not ideal, either.

The best way to structure your application is by function. Let's dissect our small application. To display the web pages, we will need a web application. We will create a single web module. Traditionally, we would say that the web application uses some services. But let's be far more careful in our naming – the web application clearly manages a cookbook. The cookbook uses the repository to store the recipes; we will implement the repository in Hibernate. We can agree that the entire application uses unified domain module.

Let's start creating the application's directory structure. We will create a Subversion repository called `s3wa`; next, we'll check it out into a working copy.

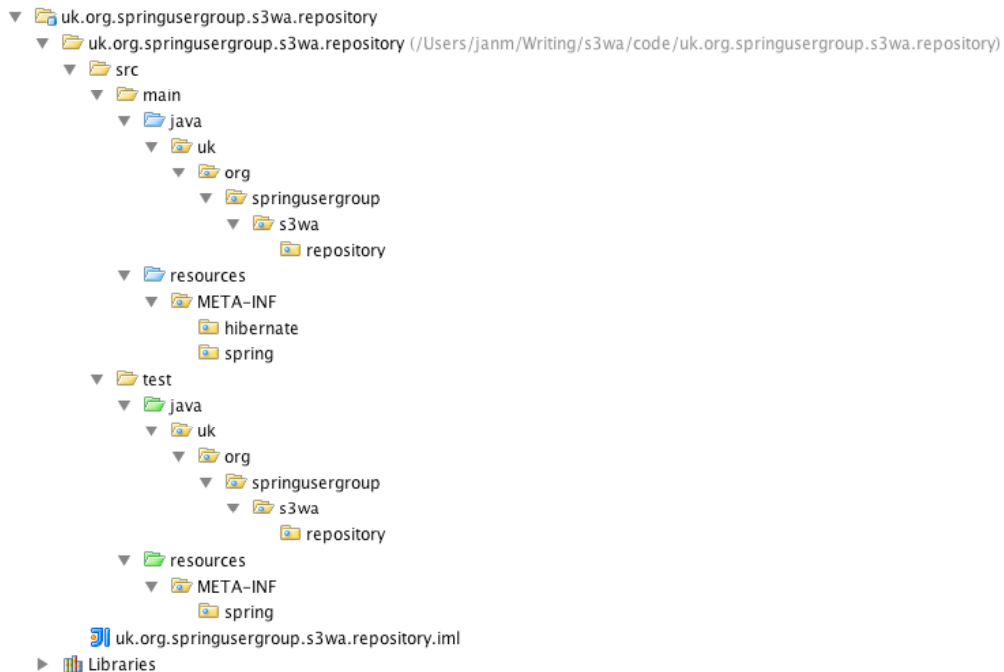
Listing 1. Creating local copy

```
./s3wa/code$ svn co https://<url>

./s3wa/code$ ls -l
total 0
```

We will create four directories that will hold the code of our modules. We'll create `uk.org.springusergroup.s3wa.domain`, `.repository`, `.cookbook`, `.web`. In each module, we are going to create directory structure to hold the source code. Figure 2 shows the directory structure of the repository module.

Figure 2. The structure of the repository module



This module, just like all others, uses this directory structure. The `src` directory holds the source code. The `main` directory contains the “production” source code, the `test` sub-directory contains source code for the tests. Each `main` and `test` directories contain at least the `java` and `resources` directories. The `java` directory contains the source code; the `resource` directory contains non-Java source code (for example the Spring XML files).

Build

Now that we have the directory structure in place, we need to think about how we’re going to build the application. Yes, you *could* use your IDE to produce the WAR, but that is a very dangerous proposition. Using your IDE means that you can only build the application on the developers’ machines; the developers may need to set up their computers in a particular way to build the application; it is usually impossible to automate the deployment to the testing platform, and the list goes on.

Unfortunately, automated build is not an easy task. In today’s Java world, you have two choices. Maven and Ant. Maven is a complex release management system, but we found that it can be a bit too heavy handed for smaller applications; it is downright unruly in large applications. We favour lighter approach using Ant and Ivy. Even though Ant is not complex, the build infrastructure needs to perform a large number of tasks, which makes it rather complex.

It is clear that our projects would benefit from a build framework. The Spring Framework provides an excellent base for such build framework. It is called spring-build; we have adapted it into cake-build, which you can download at community.cakesolutions.net. Listing 2 shows how to add reference to cake-build to our project.

Listing 2. Adding reference to cake-build

```
(s3wa) $ svn propedit svn:externals code #A

(Emacs)
cake-build https://hotei.devcake.co.uk/svn/cake-build/trunk/ #B
[C-x C-c] #C

(s3wa) $ svn update #D

Fetching external item into 'code/cake-build'
A code/cake-build/aspect
A code/cake-build/aspect/default.xml
A code/cake-build/aspect/publish.xml
A code/cake-build/aspect/common.xml
...
```

Let's explore these commands in detail. On line #A, we will modify the `svn:externals` property on the `code` directory, editing the property will launch the Emacs editor¹, line #B directs Subversion to fetch the repository at <https://hotei.devcake.co.uk/svn/cake-build/trunk> into the `cake-build` directory in the `code` directory. To exit Emacs, we must press the magic key sequence `C-x C-c`. Finally, we run `svn update` on line #D to fetch the external dependency.

Jumpstarting cake-build

The full discussion of cake-build is beyond the scope of this chapter, but we will show you how to jumpstart the cake-build. In line with the Spring Framework, we will create another directory, in `s3wa/code` called `build-s3wa`. In there, we will place the main Ant build files (see Listing 3).

Listing 3. Files in build-s3wa

```
(s3wa/code) $ ll build-s3wa
total 32
-rw-r--r--@ 1 janm staff 729 3 Apr 14:04 build.xml
-rw-r--r--@ 1 janm staff 153 15 Jan 17:03 package-bundle.xml
-rw-r--r--@ 1 janm staff 1159 15 Jan 17:03 package-top-level.xml
-rw-r--r--@ 1 janm staff 169 15 Jan 17:03 publish-top-level.xml
```

The most important file is the `build.xml` file. It defines the modules that make up our application. Let's examine its contents in Listing 4.

¹ I apologise unreservedly to the true Emacs gurus. To my humble self, Emacs is just e hellish editor. Besides, true experts write Java EE applications using `cat >*.java`.

Listing 4. The build.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="CBT" default="precommit">

  <path id="bundles">
    <pathelement
      location="${basedir}/../uk.org.springusergroup.s3wa.domain" />    #A
    <pathelement
      location="${basedir}/../uk.org.springusergroup.s3wa.repository" />#B
    <pathelement
      location="${basedir}/../uk.org.springusergroup.s3wa.cookbook" />  #C
    <pathelement
      location="${basedir}/../uk.org.springusergroup.s3wa.web" />      #D
  </path>

  <property file="${basedir}/../build.properties"/>
  <import file="${basedir}/package-top-level.xml"/>
  <import file="${basedir}/publish-top-level.xml"/>
  <import file="${basedir}/../cake-build/multi-bundle/default.xml"/>

  <target name="precommit" depends="clean, clean-integration, jar"/>

</project>
```

The lines #A ... #D tell cake-build which modules make up the application. Notice also that the order in which we specify the modules drives their dependencies. The `uk.org.springusergroup.s3wa.domain` does not depend on any other modules in this project, the `uk.org.springusergroup.s3wa.repository` *may* depend on the domain module, the `cookbook` module *may* depend on both domain and repository, and so on.

If we run `ant` in the `build-s3wa` directory now, it will fail. It will complain that there is no `build.xml` in `uk.org.springusergroup.s3wa.domain`. Even though we told cake-build that there is the `domain` module, it does not know what to do with it. In fact, we must create the `ivy.xml`, `template.mf` and `build.xml` files for every module we specified in the `build-s3wa/build.xml` file.

The `ivy.xml` defines the dependencies of the module, the `build.xml` defines the additional details about module's build process and finally, the `template.mf` is a template for the `META-INF/MANIFEST.MF` descriptor. In the OSGi world, the `META-INF/MANIFEST.MF` is crucial element of a bundle.

Now, let's take a look at the contents of these three files in the `domain` module, all in Listing 4.

Listing 4. The build.xml, ivy.xml and template.mf

```
build.xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="uk.org.springusergroup.s3wa.domain">

  <property file="${basedir}/../build.properties" />
  <property file="${basedir}/../build.versions" />
  <import file="${basedir}/../build-s3wa/package-bundle.xml" />
  <import file="${basedir}/../cake-build/standard/default.xml" />
```

```

</project>

ivy.xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="http://ivyrep.jayasoft.org/ivy-
doc.xsl"?>
<ivy-module
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "http://incubator.apache.org/ivy/schemas/ivy.xsd"
  version="1.3">

  <info organisation="uk.org.springusergroup" module="${ant.project.name}">
    <ivyauthor name="janm"/>
  </info>

  <configurations>
    <include file=
      "${spring.build.dir}/common/default-ivy-configurations.xml"/>
  </configurations>

  <publications>
    <artifact name="${ant.project.name}" type="jar" ext="jar"/>
    <artifact name="${ant.project.name}-sources" type="src" ext="jar"/>
  </publications>

  <dependencies>
    <dependency org="org.junit"
      name="com.springsource.org.junit"
      rev="${org.junit}"
      conf="test->runtime"/>
    <dependency org="org.hamcrest"
      name="com.springsource.org.hamcrest.core"
      rev="${org.hamcrest}"
      conf="test->runtime"/>
    <dependency org="org.hamcrest"
      name="com.springsource.org.hamcrest"
      rev="${org.hamcrest}"
      conf="test->runtime"/>
  </dependencies>

</ivy-module>

template.mf
Bundle-SymbolicName: uk.org.springusergroup.s3wa.domain
Bundle-Name: S3WA domain
Bundle-Vendor: Spring User Group UK
Bundle-ManifestVersion: 2

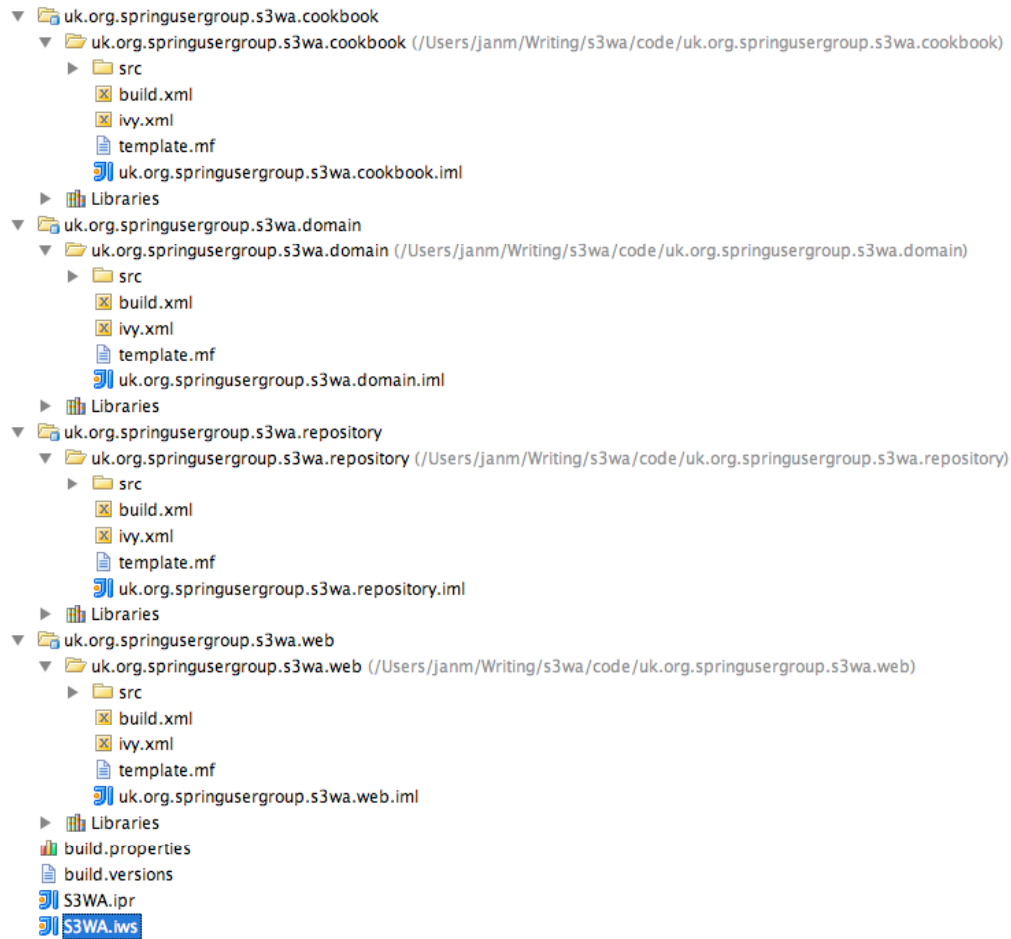
```

There will be more on ivy here

But not just yet.

Next, we need to create these three files in all modules (we can make copies, making slight changes only to `build.xml` and `template.mf`). Figure 3 shows the final view of the source code.

Figure 3. Project's source code in IntelliJ IDEA



We can now run `ant` in `code/build-s3wa`. Ant will use Ivy to fetch all required dependencies and build the project. You will see output similar to Listing 5.

Listing 5. Output from running ant in build-s3wa

```
Buildfile: build.xml

clean:
  [subant] Entering directory:
```

```

/Users/janm/Writing/s3wa/code/uk.org.springusergroup.s3wa.domain
...
jar:
  [subant] Entering directory:
/Users/janm/Writing/s3wa/code/uk.org.springusergroup.s3wa.domain

ivy.init:

resolve.init:

resolve.compile:
[ivy:cache] :: Ivy 2.0.0 - 20090108225011 :: http://ant.apache.org/ivy/
::
:: loading settings :: file = /Users/janm/Writing/s3wa/code/cake-
build/common/ivysettings.xml

compile.init:
[ivy:cache] downloading
s3://repository.springsource.com/ivy/bundles/release/org.springframework.bu
ild/org.springframework.build.ant/1.0.2.RELEASE/org.springframework.build.a
nt-sources-1.0.2.RELEASE.jar ...
[ivy:cache] ... (9kB)
[ivy:cache] (9kB)
[ivy:cache] .. (0kB)
[ivy:cache] (0kB)
[ivy:cache] [SUCCESSFUL ]
org.springframework.build#org.springframework.build.ant;1.0.2.RELEASE!org.s
pringframework.build.ant-sources.jar(src) (588ms)
[ivy:cache] downloading
s3://repository.springsource.com/ivy/bundles/release/org.springframework.bu
ild/org.springframework.build.ant/1.0.2.RELEASE/org.springframework.build.a
nt-1.0.2.RELEASE.jar ...
...

precommit:

BUILD SUCCESSFUL
Total time: 1 minute 5 seconds

```

Now that the project is built, let's examine the output. The build process placed the downloaded dependencies into `s3wa/ivy-cache/repository` and the built modules into `s3wa/integration-repo/uk.org.springusergroup`. We will configure our IntelliJ IDEA project's libraries to use the JARs in the `integration-repo` directory. This means that any developer can check out the source code from version control, run ant and open the project in his or her IDE and all will be configured.