Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Maven: Building Complex Systems

Maven is all about large-scale Java-based software projects. It advocates a standard lifecycle, standard directory layout, various best practices, extensive reports, documentation facilities, and it is totally extensible through plug-ins to boot.

April 21, 2006
URL:http://www.drdobbs.com/architecture-and-design/maven-building-complex-systems/186100398

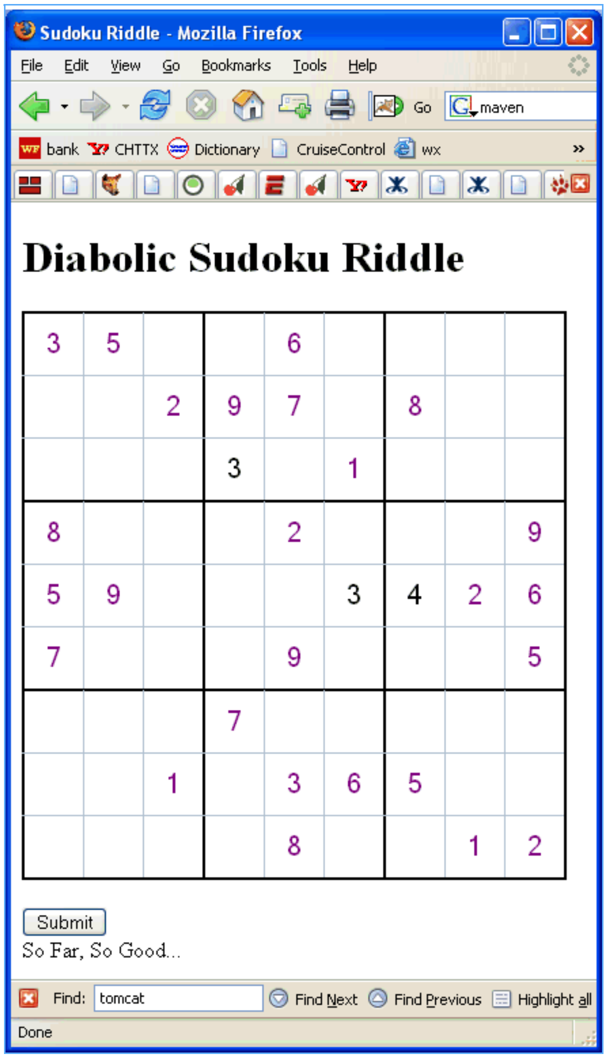
I recently had a short three-month stint on our server team. The server team works in Java and my prior Java experience included an applet that simulates broken glass and a JNI frond-end to some COM object. In other words zilch. I wasn't too worried about Java the language, since I read a lot about it and I have a good C# track record. However, the environment and the culture were very different than what I am used to (low-level C/C++). There is a lot of open source third-party code, lots of configuration files, and due to the distributed nature of the system it is not easy to perform isolated integration tests. You don't debug much in Java. You read log files and develop theories about your bugs. Consequently, I immediately volunteered to redo the build system in addition to my development tasks. Being the build guy is a great way to study a complicated software system.

The build system was a mix of ant scripts and makefiles. The system was comprised of multiple projects such a non-standard presentation tier (web tier) based on XMPP, EJB-based middle tier, and a database. In addition a panoply of satellite projects (various batch processes) were also part of the build. The middle tier was deployed on the not so prevalent JOnAS application server and the presentation tier was deployed on Apache Tomcat.

The build was far from perfect. There was no repeated one-click build, no automated testing, and deployment required manual copying of files and manual changes to various configuration files. Dependency management was pretty weak also. I studied the Java build scene and ascertained Ant is indeed the 800-pound gorilla. However I also read a lot of criticism about Ant and looked for alternatives. I finally homed in on Maven. Maven is more than a build system—it is "a software project management and comprehension tool" according to the Maven developers.

In this article, I concentrate on the build aspects of Maven. 2.0.2 (the current stable version). I introduce Maven through a web application that serves Sudoku puzzles. (The complete source code for the puzzle is available online.) The goal of a Sudoku puzzle is to populate a 9x9 grid made of nine 3x3 regions with the digits 1 through 9. Each row, column, and region should contain all 9 digits (so no repeats are allowed). You start with some populated cells and complete the rest (see Figure 1).

[Click image to view at full size]


Sudoku Riddle - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
maven
bank CHTTX Dictionary CruiseControl wx
Submit
Find Next Find Previous Highlight all
Done

Diabolic Sudoku Riddle

3	5			6				
		2	9	7		8		
			3		1			
8				2				9
5	9				3	4	2	6
7				9				5
			7					
		1		3	6	5		
				8			1	2

So Far, So Good...

Find: tomcat

Figure 1: Sudoku riddle.

What Is Maven?

Maven is all about large-scale Java-based software projects. It advocates a standard lifecycle, standard directory layout, various best practices, extensive reports, documentation facilities, and it is totally extensible through plug-ins to boot. You may, of course, customize the default directory layout and lifecycle if you have special needs. The maven core is an engine scans your project directory tree for pom.xml files and executes the build lifecycle based on the information in the pom.xml.

Maven advocates using a standard directory layout for your source files, resources, and intermediate artifacts. The benefit of this convention is that developers feel right at home when they browse a new project and don't have to specify a lot of information because Maven assumes the files are organized according to the recommend directory layout. You may override everything of course, but usually it's counter productive. Table 1 contains the standard directory layout for a single project.

Files	Explanation
src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/assembly	Assembly descriptors
src/main/config	Configuration files
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/site	Site

LICENSE.txt	Project's license
README.txt	Project's readme
target	

Table 1: Directory layout for a single project.

Maven promotes a standard build lifecycle where various plugins may hook into in order to customize or enhance it. The standard lifecycle phases are described in Table 2. The <packaging> element in the pom.xml file determines the build lifecycle for a specific project. When building a project using a command such as "mvn install" Maven will go through all the lifecycle phases up to (and including) the specified phase. The standard lifecycle makes the whole process much simpler because you don't have to graft it into the build system yourself. The good people of Maven put all the lifecycle phases that yo are likely to need even in a complex project (source-code generation, for instance) and if you happen to be an unlikely person then you are free to customize the lifecycle and hook into it using custom plugins.

Files	Explanation
validate	validate the project is correct and all necessary information is available
generate-sources	generate any source code for inclusion in compilation.
process-sources	process the source code, for example to filter any values.
generate-resources	generate resources for inclusion in the package.
process-resources	copy and process the resources into the destination directory, ready for packaging.
compile	compile the source code of the project.
process-classes	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
generate-test-sources	generate any test source code for inclusion in compilation.
process-test-sources	process the test source code, for example to filter any values.
generate-test-resources	create resources for testing.
process-test-resources	copy and process the resources into the test destination directory.
test-compile	compile the test source code into the test destination directory.
test	run tests using a suitable unit testing framework. should not require the code be packaged or deployed.
package	take the compiled code and package it in its distributable format, such as a JAR.
integration-test	process and deploy the package if necessary into an environment where integration tests can be run.
verify	run any checks to verify the package is valid and meets quality criteria.
install	install the package into the local repository, for use as a dependency in other projects locally.
deploy	done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects..

Table 2: Standard lifecycle phases.

The Project Object Model

Everything starts with the project object model (POM). The POM is defined in a pom.xml file that contains all the information necessary to build the project, generate reports and configure plugins. Maven supports the notion of a single artifact per project. This artifact may be a JAR, WAR, EAR, POM (just a pom.xml file) or some other special artifact. Listing One contains the pom.xml for the **SudokuSolver** module of the sample application. I provide very little information: parent, packaging (JAR), and dependencies. How can Maven build a project with so little information? The secret is in the standard directory layout + build lifecycle. Maven assumes various resources such as test code are located in a certain place and does the right thing about it (for example, compile the test code and run the unit tests).

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
        <groupId>GigiZone</groupId>
        <artifactId>Sudoku</artifactId>
        <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>SudokuSolver</artifactId>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Listing One: pom.xml for SudokuSolver

Maven stores project artifacts and third-party dependencies in repositories. Artifacts are identified by group id, artifact id, and version. For example, the jar file spring-core-1.2.6.jar is identified by group id: **org.springframework artifact, id: spring-code and version: 1.2.6**. It is stored in the repository at **<repository root>/org/springframework/spring-core/1.2.6**. Each element in the fully qualified artifact id gets its own sub-directory. A project that depends on other project's artifacts finds them in the repository (or rather Maven finds them and makes them available at the right time). Artifacts are typically jar files, but can also be war and ear files as well as pom.xml files. There are three types of repositories: The local repository on the developer machine, shared internal repositories (corporate repository), and public repositories. All repositories are basically directories with a certain structure.

Dependency management is one of the stickiest issues in software development. It's difficult to get it right and it's difficult to keep it right. It helps to have Maven around. Maven 2 resolves dependencies transitively. If A depends on B and B depends on C then you need both B and C in order to use A. However, A only needs to specify its direct dependencies (B in this case). Maven determines that C is also necessary and make it available. Maven executes multi-module builds in the right order building dependencies and installing them into the local repository before building the modules that depend on them. Also if later B is modified such that it needs D also then the POM of A doesn't need to change. This is a nice encapsulation where A just states it depends on B and it is not affected by implementation changes in B. Dependencies in Maven 2 have scope. The scope concept recognizes various dependencies are needed only in certain times during the development/deployment cycle. The best example is junit, which is necessary during test time only. It is not required to build your project and it is not required at deployment time. There are five scopes:

- **Compile scope** means the dependency is necessary to build your project.
- **Test scope** means the dependency is necessary to test your project.
- **Provided** means the dependency is necessary at runtime but will be provided by the container (Servlet and JSP APIs are provided by Tomcat, for instance).
- **Runtime** and **System** are esoteric scopes and I couldn't figure out a proper use case for them. The Maven documentation is pretty sketchy about it.

Plugins are Maven's middle name. The core engine runs plugins to perform its duties. Each and every build phase is executed by plugins. They are also the one true way to extend and customize Maven. A plugin is basically a set of goals, a.k.a. MOJOs (Maven Old Java Objects) that are bound to a specific lifecycle phase. Maven executes the appropriate mojo for each phase in the build lifecycle as it chugs along the build. The mojos get their input from the POM.

Who Needs Maven?

Anyone who works on a project with more than a couple of modules and more than one developer would benefit from Maven. Even if you work alone on a small project, Maven can help clean up your build act.

I am a great admirer of standards for arbitrary things. Think about curly braces, indentation, and whitespace in most languages. Who cares, right? Still lots of people have to read other people's code in unfamiliar style, or write in a style they don't like due to coding guidelines (not to mention the time and arguments to write the coding guidelines document). Maven dictates a standard directory layout, standard build lifecycle, and standard goal names. It cancels at least one huge 4 hours project meeting where everybody argues if source files should be placed under "src" or "sources" and if the output directory should be called "build", "Build", or "output". The motto of Maven is "There is one way to do it". You can have your way (and eat it too) if must, but it will cost you (in simplicity).

Maven prevents artifact and third-party dependencies duplication by storing them in the repository. This saves space and time especially if the alternative is to store all the duplicates in your SCM system. The resources in the repository should never be stored in your SCM because your build artifacts are generated from your source code (which you do store in SCM I should hope) and the third-party dependencies are never changed (and if they do then the filename is changed too to reflect it) so there is no point in storing them in a versioning system.

The project management features of Maven are its extensive support for reports, site creation, and integration with continuos integration systems. If you manage a multi-developer large-scale project, you definitely need these features.

Maven At Work

To illustrate how you use Maven, I present it here in the context of the sample SudokuServer application. The application consists of two projects and a parent pom.xml file that ties the multi-module build together. The SudokuSolver project generates a .JAR artifact and the SudokuServer project generates a .war artifact. SudokuServer depends on SudokuSolver as well as on a bunch of third-party packages.

The pom.xml files of SudokuSolver and SudokuServer (Listing One and Listing Two, respectively) inherit from the pom.xml file (Listing Three) in the root directory. This lets you put all the common information and metadata in one place. The inheriting POM files specify a <parent> element that identifies the parent POM in the repository by group id, artifact id and version. The POM inheritance mechanism is not directly related to multi-module builds.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
        <groupId>GigiZone</groupId>
        <artifactId>Sudoku</artifactId>
        <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>SudokuServer</artifactId>
  <packaging>war</packaging>
  <dependencies>
    <!-- Internal dependency on another module -->
    <dependency>
      <groupId>GigiZone</groupId>
      <artifactId>SudokuSolver</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>


    <!-- 3rd party dependencies available from the public repository -->
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.0.4</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.9</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>1.2.6</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>1.2.6</version>
      <scope>compile</scope>
    </dependency>

    <!--
      SUN dependencies that must be cannot be distributed from public respositories
      and should not be deployed to the web container (hence the scope is 'provided')
    -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.0</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.1.2</version>
      <scope>compile</scope>
    </dependency>

    <!-- Test only depdendency (should not be deployed, hence the scope is 'test') -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

  </dependencies>
</project>
```

Listing Two: pom.xml for SudokuServer.

The POM of SudokuSolver is simple as it gets. It specifies its <parent>, its <artifactId> and its <dependencies>. The group id and version are inherited from its <parent> and the <packaging> defaults to "jar". The dependencies include only junit for testing purposes (note the "test" scope). SudokuSolver has unit tests in the src/test/Java directory, but thanks to the standard directory layout there is no need to specify it explicitly. Maven finds them and invoke the tests on its own accord.

The POM of SudokuServer is a little bit more involved. This component generates a .war file and it depends on quite a few components. In addition to <parent>and <artifactId> it specifies a <packaging> element ("war") and the <dependencies> element contains dependencies with different scopes. SudokuServer depends on SudokuSolver and a bunch of third-party modules (scope: compile). It also depends on Sun's servlet-api, jsp-api and jstl jars, but they will be provided by the web container (Tomcat in this case) hence the scope is "provided". Finally, it depends on junit for testing purposes.

The root POM provides common metadata used by both modules. It specifies a <packaging> element ("pom"), so its artifact is the pom.xml file itself, which is stored in the repository so inheriting projects can locate it. It specifies also a <url> and <developers> elements that can be used for documentation and site generation.

Multi-module Builds

The POM (Project Object Model) is kind of a misnomer because in multi-module projects every module will have its own POM in a separate pom.xml file. I guess MOM (Module Object Model) sounds silly to tough and brawny Maven developers. The documentation mentions a super POM, but I couldn't figure out if they mean the set of all elements that may appear in a POM or the top-level POM so I will not use the "super POM" term. The top-level POM file (see Listing Three) specifies a <modules> element that defines the build order. When Maven encounters a <modules> element it drops whatever it's doing and starts building the modules. You may have <modules> elements in intermediate POMs too and create a controlled cascading build. The build order is: SudokuSolver and then SudokuServer. It makes sense since SudokuServer depends on SudokuSolver, so SudokuSolver must be built first. The dependency resolution mechanism doesn't resolve this case automatically since both modules are siblings.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>GigiZone</groupId>
  <artifactId>Sudoku</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>Sudoku Parent POM</name>
  <url>http://localhost:8080/SudokuServer-1.0-SNAPSHOT/sudoku.html</url>
  <packaging>pom</packaging>
  <modules>
        <module>SudokuSolver</module>
        <module>SudokuServer</module>
  </modules>
  <developers>
    <developer>
      <name>Gigi Sayfan</name>
      <email>gsayfan@playstation.sony.com</email>
      <roles>
        <role>Lead Developer</role>
      </roles>
      <timezone>0</timezone>
    </developer>
  </developers>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
           <source>1.5</source>
           <target>1.5</target>
        </configuration>
     </plugin>
    </plugins>
  </build>
  <repositories>
    <repository>
      <id>Maven Snapshots</id>
      <url>http://snapshots.maven.codehaus.org/maven2/</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <releases>
        <enabled>false</enabled>
      </releases>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>Maven Snapshots</id>
      <url>http://snapshots.maven.codehaus.org/maven2/</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <releases>
        <enabled>false</enabled>
      </releases>
    </pluginRepository>
  </pluginRepositories>
</project>
```

Listing Three: Top-level POM file.

Configuring Plugins

The top-level POM file (Listing Three) specifies a <plugin> element that configures the compiler plugin to use Java 1.5 source compatibility. The default is 1.3 for backward compatibility reasons. This mechanism allows configuring any plugin (builtin or custom) in a simple and consistent way. just add more plugin elements, identify them by group and artifact id and add the information to the <configuration> element of each plugin.

Maven's built-in plugins are not the end of the road. Many interesting plugins are available from different sources. I show you how to use the Tomcat plugin from codehaus. The Tomcat plugin resides in the Subversion repository of the codehaus. You need to install Subversion, get the source of the plugin build it and install it into your Maven's plugin repository.

Use this command to get the source:

```
svn checkout https://svn.codehaus.org/mojo/trunk/mojo/mojo-sandbox/tomcat-maven-plugin tomcat-maven-plugin
```

Once you got the source just go to the tomcat-maven-plugin directory and enter:

```
mvn install
```

The plugin will be installed and as long Tomcat is installed properly can deploy your app by typing:

```
mvn deploy
```

The tomcat plugin offers many goals for managing the Tomcat server: start, stop, list, info, and so on. Using the plugin require a special and elements in the top-level POM file since this plugin is not distributed through the standard ibiblio repository where it can be found automatically.

Maven Versus Ant

Maven is the evolutionary successor of Ant. You can directly execute ant tasks in Maven (through a plugin of course), so Maven is a superset. However, this is not the whole story. The conceptual model is different. Ant is all about total freedom, which is not always good. Maven gives structure, advises on best practices and tries to solve some thorny problems like dependency management and shared artifacts in a neat way. If your project build deviates from Maven so much that it's easier to do it in Ant, I recommend you take a serious look and figure out why. Ant has the advantage of being mature, familiar and better documented. All are fleeting advantages and I belive Maven will gain more market share as time goes by.

Maven 1.x Versus Maven 2.x

Maven 2 is a radical departure from Maven 1. A lot has changed and many people belive non-compatible Maven 2 will hurt Maven's adoption in the short range. I agree, but I still think it's better in the long run. Maven 2 introduced hierarchical repositories. you can rely on some structure and organization in order to find something instead of plowing through a huge flat list of artifact directories. Maven 1 didn't have transitive dependencies, so you had to specify in every project all the dependencies of this project and all its dependencies' dependencies ad infinitum. Every dependency change in a POM down the line required fixing the POMs of all the projects that depend on it directly or indirectly. It becomes a maintenance problem pretty soon in big projects. Maven 1 didn't have the notion of scoped dependencies so you ended up either deploying junit and various jars provided by the web container or had to hack around your maven.xml file. Maven 1 plugins where mostly developed using Jelly—an XML-eque scripting language. It wasn't the best vehicle for developing plugins. Maven 2 plugins are developed using Java.

IDE Support

There is rudimentary support only for Maven 2. There are plugins for popular IDEs like Eclipse, NetBeans, IDEA, and JBuilder. They mostly let you generate IDE projects from POMs and edit the POM in the IDE instead of in raw XML.

bTroubleshooting Build Problems

Builds go wrong. Especially, when create a new build system or change something major (for instance, add a new type of artifact or life cycle phase). This is one of the weak points of Maven (and any other build system out there with the possible exception of SCons). There is very little material on the web on troubleshooting build problems. Maven's **-e** and **-X** flags are your friends. These flags tell maven to spew verbose information that may help you to glean some insights.

SudokuServer: A Sample Web Application

The sample application I present is a minimalist MVC web application that consists of a simple model component (*SudokuSolver*) and a view/controller web component (*SudokuServer*). The application serves a hard-coded Sudoku puzzle and allows the user to check for partial solutions. I will not dwell on the implementation much since there is nothing spectacular there, it just serves a as vehicle to demonstrate Maven.

I used Spring the web/infrastructure framework for the web. The *spring-webmvc* module provides various features such dispatcher servlet handler mappings, view resolution, JSP tag lib, and the like. It is very configurable and flexible. There is no connection or any support between Maven and Spring. Any web framework would do for this application.

The application components include:

- SudokuSolver. Contains the *Board* class. This is a POJO that wraps a Sudoku board. It is used to serve the initial riddle to the user as well as to carry back the user's partial/full solution. The *Board* can check itself and determine if it is complete, partial or contains an error.
- SudokuServer: Contains the *SudokuServerController* and *SudokuServerValidator*. These classes control the workflow of requests and determine the response sent back to the user (via jsp+css). There is no significant JavaScript and it's definitely not an AJAX application. (There. I did it. Now, a search for AJAX will turn up this article.)

The directory structure is the standard Maven directory structure. The web component (SudokuServer) has some web-specific sub directories like WEB-INF and META-INF. The model component (SudokuSolver) has a test sub-directory. Listing Four contains the exact directory layout with all the input files. These files include: Java source files, various XML descriptors, JSP templates and Maven's build files. It seems like a lot of structure and metadata for such a simple project and it really is. However, keep in mind that the same structure is supposed to support huge large-scale projects with hundreds or thousands of modules.

```
SudokuServer-o
             |
            src-o
             |  |
             | main-o
             |      |
             |     java-o
             |      |   |
             |      |  com-o
             |      |      |
             |      |  GigiZone-o
             |      |           |
             |      |      SudokuServer-o
             |      |                    |
             |      |        SudokuServerController.java
             |      |        SudokuServerValidator.java
             |      |
             |     webapp-o
             |            |
             |           META-INF-o
             |            |       |
             |            |      context.xml
             |            |
             |           WEB-INF-o
             |            |       |
             |            |      jspf-o
             |            |       |   |
             |            |       |  include.jspf
             |            |       |  sudoku.jspf
             |            |       |
             |            |  SudokuServer-servlet.xml
             |            |  spring.tld
             |            |  web.xml
             |            |
             |        index.jsp
             |        sudoku.css
             |
            pom.xml

SudokuSolver-o
             |
            src-o
             |  |
             | main-o
             |  |    |
             |  |  java-o
             |  |      |
```

Listing Four: Directory layout.

The interesting thing about the build files is that they don't reflect the complexity of the directory layout. The build files contain exactly the information necessary to build the project, which is more or less identify the artifact, configure the build with various plugins and state the dependencies of each module. This is a nice demonstration of the "convention over configuration" approach. The only thing I would add is for the test project to depend by default on junit since it is so prevalent and the de-facto standard for Java unit testing.

Building, Deploying, and Running the SudokuServer

SudokuServer requires Apache Tomcat be installed on your system. Building and deploying is as simple as typing 'mvn deploy' at the command line. All the preparation work on the build files now pays off. Maven will compile the sources, package the various artifacts in jars, run the tests and finally will deploy it into Tomcat. If you wish to extend this project with additional modules or more files in existing modules the build infrastructure is ready for you. Running is as simple as pointing your browser to http://localhost:8080/SudokuServer-1.0-SNAPSHOT/. SudokuServer always serves the same hard-coded riddle at the moment for simplicity and ease of testing. Future versions will generate new riddles automatically and/or scrape the web for new ones. Have fun.

DDJ

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
        <groupId>GigiZone</groupId>
        <artifactId>Sudoku</artifactId>
        <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>SudokuSolver</artifactId>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

```
<!-- 3rd party dependencies available from the public repository -->
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.0.4</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.9</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>1.2.6</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>1.2.6</version>
      <scope>compile</scope>
    </dependency>

    <!--
      SUN dependencies that must be cannot be distributed from public respositories
      and should not be deployed to the web container (hence the scope is 'provided')
    -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.0</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.1.2</version>
      <scope>compile</scope>
    </dependency>

    <!-- Test only depdendency (should not be deployed, hence the scope is 'test') -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>GigiZone</groupId>
  <artifactId>Sudoku</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>Sudoku Parent POM</name>
  <url>http://localhost:8080/SudokuServer-1.0-SNAPSHOT/sudoku.html</url>
  <packaging>pom</packaging>
  <modules>
        <module>SudokuSolver</module>
        <module>SudokuServer</module>
  </modules>
  <developers>
    <developer>
      <name>Gigi Sayfan</name>
      <email>gsayfan@playstation.sony.com</email>
      <roles>
        <role>Lead Developer</role>
      </roles>
      <timezone>0</timezone>
    </developer>
  </developers>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <repositories>
    <repository>
      <id>Maven Snapshots</id>
      <url>http://snapshots.maven.codehaus.org/maven2/</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <releases>
        <enabled>false</enabled>
      </releases>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>Maven Snapshots</id>
      <url>http://snapshots.maven.codehaus.org/maven2/</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <releases>
        <enabled>false</enabled>
      </releases>
    </pluginRepository>
  </pluginRepositories>
```

```
             |  |      com-o
             |  |         |
             |  |      GigiZone-o
             |  |               |
             |  |            Sudoku-o
             |  |                   |
             |  |              Board.java
             |  |
             | test-o
             |    |
             |   java-o
             |       |
             |     com-o
             |         |
             |      GigiZone-o
             |                 |
             |              Sudoku-o
             |                     |
             |                    Test-o
             |                         |
             |                   BoardTest.java
             |
            pom.xml
pom.xml
```

Listing One: pom.xml for SudokuSolver

Listing Two: pom.xml for SudokuServer.

Listing Three: Top-level POM file.

```
</project>
```