

BENEFITING FROM OPEN SOURCE DEVELOPMENT

by Sumitra Chary, Christian Donner,
Jim Lamoureux, Ilia Papas, and Dita Vysloulzil

*The goal: cross-platform
Java development*

In a market that is defined by today's tight IT budgets, saving on software licenses can mean the difference between financial failure and success for a software development project. While our corporate clients use commercial-grade application servers, we sometimes find ourselves in a situation where there are no funds for developer licenses of these commercial application servers. Out of necessity, we developed and implemented a process that allows for development on top of an open source stack, while production delivery relies on a commercial application server.

Initial concerns that implementation differences and the different runtime environments would lead to issue-prone deployments turned out to be unjustified. While different application servers do indeed show incompatibilities, we found that we were able to avoid common pitfalls through preparation and disciplined coding. In this article, we will explain what it takes to develop complex Web applications with Eclipse and Tomcat and to deploy these applications to a WebSphere-based production environment.

Introduction

It all started when a client requested a solution for the WebSphere application server platform, but did not want to cover the cost of WebSphere Studio licenses for the development team. We looked for alternatives and found one in Eclipse and Tomcat.

The team initially feared that the different implementation of core functionalities provided by application server containers would create application portability issues. The main areas of concern included transaction management, security, and application deployment.

Because we used IBM's Tivoli Access Manager and WebSEAL Reverse Proxy in production, but relied on Tomcat's built-in authentication in development, there was concern that having only a subset of the target security infrastructure available in development would limit our ability to build a security service layer for Tivoli.

These risks had to be addressed and dealt with. At that time it seemed that the cost of doing so would outweigh the potential savings from software licenses. However strong this concern was, it was difficult to convey it to a client who was eager to start the project, and so we embarked on the open source endeavor.

Developing with Eclipse and Tomcat

Once properly configured, Eclipse can be a powerful hub for developing your application. It can automatically generate content and code such as class header comments, implementations of functions from interfaces, variable getters and setters, and more. These time-saving tools, along with the multitude of available plug-ins (e.g., for Tomcat, VSS, and Struts) allowed us to spend less time performing repetitive tasks and more time actually developing.

We created a project in Eclipse with its root reflecting the root of our Web application, which would later be packaged into a WAR (Web Application Archive), then an EAR (Enterprise Archive), along with the required application configuration files, for deployment to WebSphere. This root directory was located within the "webapps" directory of our Tomcat installation, which is the default directory that Tomcat allocates for Web applications.

| | | |
|-----------------------------|-------|--|
| IBM WebSphere | 5.0.2 | WebSphere Application Server |
| MS SQL Server | 2000 | The client licensed Microsoft's database |
| IBM JCK | 1.3.1 | Required, the application does not work with Sun's JCK |
| Jakarta Struts | 1.2.4 | Open Source MVC Framework |
| Spring | 1.1.1 | Open Source Framework |
| JTDS | 1.0 | Open Source SQL Server JDBC driver |
| Tivoli Access Manager | 5.1 | Tivoli Access Manager for eBusiness (TAMeb) |
| IBM Tivoli Directory Server | 5.2 | LDAP Server |
| IIS | 5.0 | Microsoft's Internet Information Service |

Table 1 Development Stack

| | | |
|-----------------------|--------|--|
| Eclipse | 3.0.1 | Open Source IDE |
| Jakarta Tomcat | 5.0.28 | Open Source Application Server |
| Jakarta Struts | 1.2.4 | Open Source MVC Framework |
| Eclipse VSS Plugin | 1.6.0 | Source control plugin for Eclipse and MS Visual Source Safe |
| Eclipse Tomcat Plugin | 3.0 | Start, stop and recycle Tomcat from within the IDE |
| Jakarta Log4j | 1.2.8 | Open Source logging and tracing framework |
| MS SQL Server | 2000 | This is optional. Developers can use a shared database server instead of deploying the database locally. |
| IBM JCK | 1.3.1 | Required, the application does not work with Sun's JCK |
| Spring | 1.1.1 | Java Framework |
| JTDS | 1.0 | Open Source SQL Server JDBC driver |

Table 2 Production Stack



Sumitra Chary is a senior software engineer at Molecular. Her career has spanned both academic and commercial worlds. These have included software systems for X-ray observatory missions, network management, marketing automation, and enterprise Web applications.

schary@moecular.com

Although the Tomcat plug-in for Eclipse does not add any new functionality to either product, it greatly eases the integration of the two and saves time by consolidating common tasks in one place and reducing the need for multitasking. Debugging in Eclipse is fairly robust, allowing the user to step through code and to evaluate expressions on the fly. The JDK we were using (IBM 1.3.1) does not support hot-replacing of classes, but new code is loaded on an application server restart, which does not take much time.

It should be mentioned that Tomcat does not support Enterprise beans. We decided against Enterprise beans because the Spring framework provides similar features without the platform dependencies.

The Microsoft Visual SourceSafe plug-in integrates well into the Eclipse interface, allowing for comments on both checkout and check-in. It also provides a report of all files checked out within the project, the owner, and what actions are being performed on them. The only gripe is that when checking-in files, it does not remember the checkout comment, so it must be reentered manually.

There are a few aspects to take into consideration when bridging the gap between the development and production environments. User authentication, handled by Tivoli Access Manager in production, was handled by the *tomcat-users.xml* file located in the *config* directory. Roles, users, and passwords are recorded in this file. Through the use of configuration files and Ant, we were able to easily change server locations and credentials, as well as any other variables that may need to change when code is

moved between environments. Tomcat tends to be much more forgiving when it comes to parsing configuration files such as the *web.xml* and tag library definitions, whereas WebSphere will either load the application in a crippled state or not at all. The *dtls* must be adhered to in order to avoid this issue.

Production Environment

The production environment was a load-balanced configuration of two application servers and several other servers hosting the security environment (Tivoli Access Manager) and the database (see Tables 1 and 2 and Figure 1).

Multiple Environments

In most software development projects, to support the life cycle of the application, there are multiple environments into which the code must be deployed (see Figure 2).

When the application is deployed from one environment to another, various things need to change, such as database data source information and LDAP server information. We used Ant's property filtering capability to generate runtime resource files, such as properties files and Spring application context files, with the correct information appropriate to each environment.

We recommend the following steps to make this work:

1. Define a *deploy.host* property and assign a value according to the hostname of the target deployment environment



Christian Donner is a senior consultant and technical architect at Molecular. He is a Certified Sun Enterprise Architect for Java 2 and devotes much of his career to helping clients integrate complex Web applications with their grown corporate IT infrastructures. Christian has 20 years of experience in software development

cdonner@molecular.com



DynamicPDF™ components will revolutionize the way your enterprise applications and websites handle printable output. DynamicPDF™ is available natively for Java.

DynamicPDF™ Merger v3.0

Our flexible and highly efficient class library for manipulating and adding new content to existing PDFs is available natively for Java

- ♦ Intuitive object model
- ♦ PDF Manipulation (Merging & Splitting)
- ♦ Document Stamping
- ♦ Page placing, rotating, scaling and clipping
- ♦ Form-filling, merging and flattening
- ♦ Personalizing Content
- ♦ Use existing PDF pages as templates
- ♦ Seamless integration with the Generator API

DynamicPDF™ Generator v3.0

Our popular and highly efficient class library for real time PDF creation is available natively for Java

- ♦ Intuitive object model
- ♦ Unicode and CJK font support
- ♦ Font embedding and subsetting
- ♦ PDF encryption ♦ 18 bar code symbologies
- ♦ Custom page element API ♦ HTML text formatting
- ♦ Flexible document templating

Try our free Community Edition!



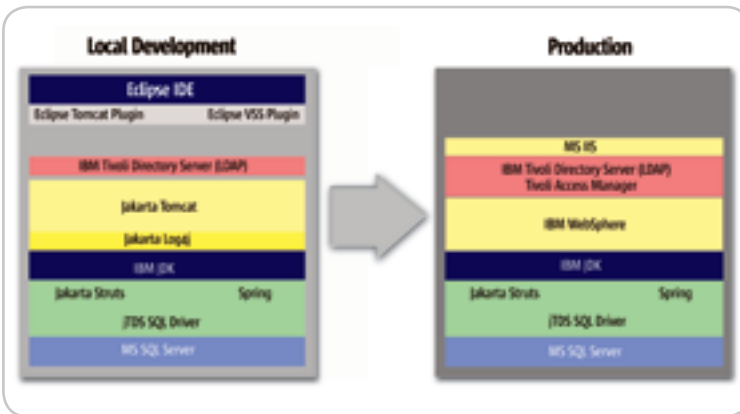


Figure 1 Development and Production Stack

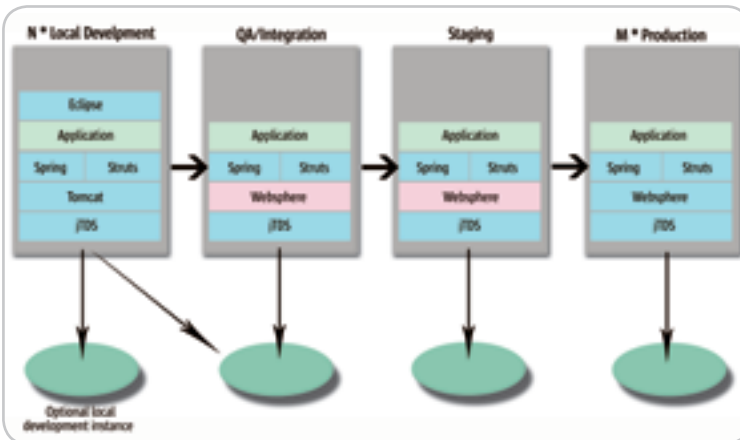


Figure 2 Multiple Environments

2. Create a separate properties file for each host with environment-specific values. For example, JDBC property definitions for *serverA* might be defined in *serverA.properties* as follows:

```
#
# Database overrides
#
jdbc.driver.classname = net.sourceforge.jtds.jdbc.Driver
jdbc.driver.type       = jtds
jdbc.server.type       = sqlserver
jdbc.server.port       = 1433
jdbc.server.host       = db01
jdbc.username          = db01-user
jdbc.password          = db01-pwd
```

3. Create Ant filter token definitions that use the environment-specific properties:

```
<filterset id="project.filter.tokens">
  <!-- DB Service(s) -->
  <filter token="JDBC.DRIVER.CLASSNAME" value="{jdbc.driver.class-
name}"/>
  <filter token="JDBC.DRIVER.TYPE"       value="{jdbc.driver.
type}"/>
  <filter token="JDBC.SERVER.TYPE"      value="{jdbc.server.
type}"/>
</filterset>
```

```
<filter token="JDBC.SERVER.HOST"      value="{jdbc.server.
host}"/>
<filter token="JDBC.SERVER.PORT"      value="{jdbc.server.
port}"/>
<filter token="JDBC.USERNAME"         value="{jdbc.username}"/>
<filter token="JDBC.PASSWORD"         value="{jdbc.password}"/>
</filterset>
```

4. Create a properties file containing the filter tokens. Ant will substitute actual values for the tokens:

```
jdbc.driverClassName = @JDBC.DRIVER.CLASSNAME@
jdbc.url = jdbc:@JDBC.DRIVER.TYPE:@JDBC.SERVER.TYPE@://@JDBC.
SERVER.HOST@:@JDBC.SERVER.PORT@
jdbc.username = @JDBC.USERNAME@
jdbc.password = @JDBC.PASSWORD@
```

5. Place a *copy* task in some target that invokes the filter token substitution (<filterset>):

```
<target name="copy-files" depends="">
  <!-- Copy, with overwrite, properties and xml files
  - so that configuration changes via Ant build properties
  - will always be picked up.
  -->
  <copy todir="{web.build.dir}" overwrite="yes">
    <fileset dir="{web.src.dir}">
      <include name="**/*.properties" />
      <include name="**/*.xml" />
    </fileset>
    <filterset refid="project.filter.tokens" />
  </copy>
</target>
```

When the application is packaged, it looks in (among other places) `{web.build.dir}` for files to include in the Web application archive (WAR). There, it will find the generated runtime resources with environment-specific values.

Spring

The Spring Framework was very useful in allowing us to develop our application in a container-agnostic fashion. We took advantage of several of the many features of Spring.

1. Service Location

We used Spring application contexts for the integration with Struts, for deployments to Tomcat and WebSphere, in standalone utility applications, and even in JUnit tests. Spring allowed us to standardize how our service objects were found and initialized across all uses of those objects in a compelling way.

2. Bean Life Cycle and Dependency Management

By using Spring's application contexts, we successfully avoided stateless session beans that would have caused deployment issues across containers (not to mention the fact that Tomcat would not have readily supported EJBs).



Jim Lamoureaux is a senior consultant and software architect at Molecular. His interests include object-oriented design and implementation, programming languages, and software process. Jim is a Sun Certified Programmer for the Java 2 Platform. He currently lives in Southern New Hampshire.

jim@molecular.com

Configuring Ant for Deployments Between Different Application Servers

We used Ant (Ant 1.6+) to manage configuration, builds, and deployments from local development environments to the integration server, from there to the staging server, and finally to production. The ant scripts needed to handle two main server differences:

1. The *WEB-INF/lib* directory had to be populated with any JARs not provided by the application server. Specifically, our Tomcat environment required the optional JDBC 2.0 Package while WebSphere already came with the necessary classes installed.
2. The security-* elements of the Web deployment descriptor (web.xml) needed to include security-role definitions for deployments to Tomcat. In WebSphere, the security roles were defined at the enterprise application level (application.xml).

The solution was to treat any environment dependencies through parameters and to create configuration files that contained all settings for a server type. We laid the groundwork by explicitly providing a value for the *server.type* Ant property:

```
<!-- Server Type property-override customizations (if any) -->
<property name="server.type.config.file" location="${build.modules.home}/
deployment/servertypes/${server.type}.properties"/>
<echo message="server.type.config.file=${server.type.config.file}"/>
<property file="${server.type.config.file}"/>
```

Having a separate properties-file for each server type was helpful, because it made the deployment process agnostic of the type of server that we deployed to. The main property set in each of these files was *deploy.tomcat* or *deploy.websphere* (essentially *deploy.server.type*). Having these properties allowed us to configure the *build-war* macro according to the server type to handle the inclusion/exclusion of the JDBC 2.0 optional package (see Listing 1).

Only one of the war-* targets is being called depending upon whether the *deploy.websphere* property is defined or not. This results in a macro definition of *build-war*, which has been configured for the target server.

Similarly simply, the appropriate definitions for the security-* elements of the web.xml are handled according to the value of *server.type*.

```
<!-- Copy the environment-specific version of the web-security.xml XDoclet
merge file -->
<target name="web-security-websphere" if="deploy.websphere">
  <copy file="${web.merge.dir}/was-web-security.xml"
    tofile="${web.merge.dir}/web-security.xml" overwrite="yes"/>
</target>
<target name="web-security-tomcat" unless="deploy.websphere">
  <copy file="${web.merge.dir}/tomcat-web-security.xml"
    tofile="${web.merge.dir}/web-security.xml" overwrite="yes"/>
</target>
```

The targets *web-security-tomcat* and *web-security-websphere* are then named as dependencies in other targets that use the XDoclet *webdoclet* task (which uses the *web-security.xml* deployment descriptor snippet).

Listing 1: Ant macro for building a WAR file

```
<!-- Call the build-war macro that is defined by the dependencies
-->
<target name="package-web"
  depends="webdoclet,war-tomcat,war-websphere">
  <build-war/>
</target>

<!-- Setup the build-war macro for a tomcat deploy -->
<target name="war-tomcat" depends="" unless="deploy.websphere">
  <macrodef name="build-war">
    <sequential>
      <war destfile="${web.dist.dir}/${web.war}"
        webxml="${web.build.dir}/WEB-INF/web.xml"
        compress="true">
        <fileset dir="${web.build.dir}" excludes="**/web.xml" />
        <webinf dir="${struts.dir}" includes="validator.
xml,*.dtd" />
        <lib dir="${cfmx.dir}" includes="*.jar" />
        <lib dir="${commons-lang.dir}" includes="*.jar" />
        <lib dir="${dist.dir}" includes="${dist.name}" />
      </war>
    </sequential>
  </macrodef>
</target>

<!-- Setup the build-war macro for a WebSphere deploy -->
<target name="war-websphere" depends="" if="deploy.websphere">
  <macrodef name="build-war">
    <sequential>
      <war destfile="${web.dist.dir}/${web.war}"
        webxml="${web.build.dir}/WEB-INF/web.xml"
        compress="true">
        <fileset dir="${web.build.dir}" excludes="**/web.xml" />
        <webinf dir="${struts.dir}" includes="validator.
xml,*.dtd" />
        <lib dir="${commons-lang.dir}" includes="*.jar" />
        <lib dir="${dist.dir}" includes="${dist.name}" />
        <lib dir="${jstl.lib.dir}" includes="*.jar" />
        <lib dir="${struts.dir}" includes="*.jar" />
        <lib file="${commons-dbc.jar}" />
        <lib file="${commons-pool.jar}" />
        <lib file="${log4j.jar}" />
        <lib file="${spring.jar}" />
        <lib file="${jdbc.jar}" />
        <lib file="${jtds.jar}" />
      </war>
    </sequential>
  </macrodef>
</target>
```

3. JDBC Template Code

The Spring JDBC APIs allowed for facile database coding – much cleaner code and standardization along the lines of connection management and exception handling.

4. Flexible Data Sources

The Spring model of using beans to wire together dependent objects allowed us to use extra-container data sources. This came with the benefit of standardized usage of data sources across our runtime scenarios – no fiddling around with container-specific data source configuration.

Jakarta Commons Logging API

We used the Jakarta Commons Logging API from the beginning. It provides a very useful abstraction of typical logging needs while supplying useful hooks for plugging in various logging services such as Log4j, the Java Logging API, etc. WebSphere even provides a gateway to its own tracing facility. The *ws-commons-logging.jar* in the lib directory off the WebSphere installation root directory allows for logging of classes to be controlled via the WebSphere Administrative Console – as long as those classes were coded to use the Jakarta Commons Logging API.

Commons Logging allowed us to configure which plugin to use (e.g., Log4J) in a Tomcat environment,

Tivoli Access Manager

The production security configuration followed the recommendations for Tivoli implementations published by IBM. The setup consisted of two WebSEAL servers, two Web/application servers, one policy server, and a master/replica LDAP configuration. The application servers hosted all of the applications with WebSEAL tying to each application through an IP/Port specific junction (a "junction" is a resource mapping and defines the true location of a URI). This necessitates multiple network cards in the WebSEAL machines in order to support multiple host addresses that are on the standard Web port.

Each production WebSEAL instance had numerous junctions configured to the multiple applications. The configuration was also set up for failover by ensuring that the server UUID configured in the junctions matched on each machine; therefore cookies for session fail-over could be picked up by either WebSEAL instance.

Choosing to install the Authorization Server on each application server created policy server redundancy. The authorization servers act as a replica of Policy server information. As a default, when the authorization server is installed, the application server does not hit the policy server directly in most cases because it obtains authorization information directly from the authorization server. The only time the policy server is reached is for any account updates. All these settings can be found in a configuration file (webseald.conf). Choosing to follow the authorization server route ensures application availability in case the policy server is down – it's a more economical method for fail-over than a master/replica policy server configuration.

WebSphere logging in that environment) and – via its default implementation that simply writes to the console – to trace unit test code without the need to configure or enable a logging service. In addition, we were able to completely turn off logging via configuration files. (This is done by placing a file called commons-logging.properties on the classpath with the line `org.apache.commons.logging.Log=org.apache.commons.logging.impl.NoOpLog` in it). In fact, this was the standard configuration for running our unit tests, which were run as part of every build. Of course, if a unit test failed, logging could be turned on again as a diagnostic tactic by setting `org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog`.

What About the Risks?

After a year of real-life experience of developing several applications and performing multiple production deployments with this configuration, we feel that developing on Tomcat and deploying to WebSphere is a low-risk strategy. Once the environments were set up and the deployment process automated, there were very few problems. Spring provided the necessary container capabilities that we needed in a portable way. We were able to take advantage of Spring's bean management, service locator, data source, and JDBC abstractions.

Differences in the security infrastructure were overcome by using Tomcat's built-in features and by providing stub code in the service layer of the development environment that simulated the presence of TAM. We were able to use the same LDAP server in development (with Tomcat security) and in QA (with Tivoli Access Manager security).

Over time, the cost-saving aspect of cross-platform development became less important in favor of other advantages that were initially not anticipated. The lightweight development environment turned out to be a great advantage, and being forced to layer the application architecture to achieve isolation from the container produced cleaner and better maintainable application code – something that reduced the overall project risks, not increased them.

Summary

It takes a good amount of planning to develop on Tomcat and successfully deploy to a WebSphere environment. Open source frameworks, such as Spring and Struts, can be used to shield an application from platform-dependent implementation details. Ant is a handy tool that facilitates cross-platform deployments. Special consideration is required to handle application security across different platforms. Coding guidelines designed to avoid platform-dependencies must be followed rigorously.

With all these things in mind, cross-platform Java development is a rewarding goal, because your resulting application will be cleaner, easier to maintain, and can provide a real cost advantage. ☺

Resources and Links

- *JDBC package for Tomcat with JVM 1.3.1*: <http://java.sun.com/products/jdbc/articles/package2.html>
- *IBM WebSphere*: <http://www.ibm.com/developerworks/websphere>
- *IBM Tivoli Access Manager*: <http://www.ibm.com/developerworks/tivoli>
- *Struts*: <http://struts.apache.org>
- *Spring Framework*: <http://www.springframework.org>
- *Commons Logging*: <http://jakarta.apache.org/commons/logging>
- *jTDS JDBC Driver*: <http://jtds.sourceforge.net/>
- *Info Center for Tivoli – with related replication/fail-over configurations*: http://publib.boulder.ibm.com/infocenter/tivohelp/v2r1/index.jsp?toc=/com.ibm.itame.doc_5.1/toc.xml
- *Great detailed intro to Tivoli Access Manager – must read for anyone considering TAM*: <http://www.redbooks.ibm.com/redpapers/pdfs/redp3677.pdf>



Ilia Papas is a software engineer at Molecular. He has been working with web applications for five years and has interests in the design and implementation of enterprise applications using a variety of technologies. He currently lives in the Boston area.

ipapas@molecular.com



Dita Vysloulzil is a Consultant and Technical Architect in the Engineering group at Molecular in Watertown. She has been in software development for 7 years, concentrating in transactional web applications.

dvysloulzil@molecular.com

“Once properly configured, Eclipse can be a powerful hub for developing your application”