Migrating from WebLogic 8.1 to 9.0 - A case study

By Thomas Kruse and Alois Lechicki
August 2006

Abstract

Migrating to new versions of an application container can be challenging, but on the other hand it allows one to keep pace with product development and improvement.

Moving from WebLogic 8.1 to 9.0 involves some issues, e.g. you have to make your application compatible with JDK 1.5.

Here are some practical tips to make the migration working based on our experience gained in a large-scale J2EE project.

Introduction

Migrating to new versions is always risky, but is done for efficiency and new capabilities, and to stay up to date with a vendor's products and support. Whether you are currently using WebLogic 8.x or an earlier version, there are many reasons that justify the move to WebLogic 9.x. By moving to WebLogic 9.x, you can take advantage of all its enhancements and new features (see [4, 5]), which should match your growing business and technical needs.

As WebLogic Server 9.0 was released a few months ago, the IT department of a travel retailer decided to migrate a large-scale J2EE system running on the WLS8.1 SP3 platform to WebLogic 9.0. The system in question was a Web-based selling portal for travel agents (see next section for details). It was decided to carry out the migration during development of a next release of the system.

BEA published an excellent guide entitled *Upgrading WebLogic Application Environments* ([2]) and provided an *Upgrade Wizard* which supports the migration process. While the guide provides step-by-step instructions on how to migrate your application environment, there are no tools for automating the process of upgrading configurations and build & deployment processes.

As a result, IT departments are forced to do the migrations to a large extent manually. The following sections describe a real-life example of a WebLogic migration and provide some practical tips on how to make this type of migration work.

Application Overview

The application HolidayTrader we will refer to in this article is a Web-based selling platform offering travel agents direct access to information and reservation systems of a wide range of travel providers (see [7] for more details).

The system had already been in production for two years and was used by thousands of travel agents in Germany. Agents could select from over 100 million package holidays and over 10 million last-minute travel bargains, updated daily. The system enabled users to search for a desired holiday package, check room and flight availability, then book and track customer reservations.

The HolidayTrader is ideal for the purpose of our migration walkthrough since it utilized most areas of J2EE functionality: WebServices, Java Server Pages (JSP), servlets, and EJB components, as well as messaging via the JMS, and database access with JDBC. Figure 1 depicts an overview of the HolidayTrader architecture.

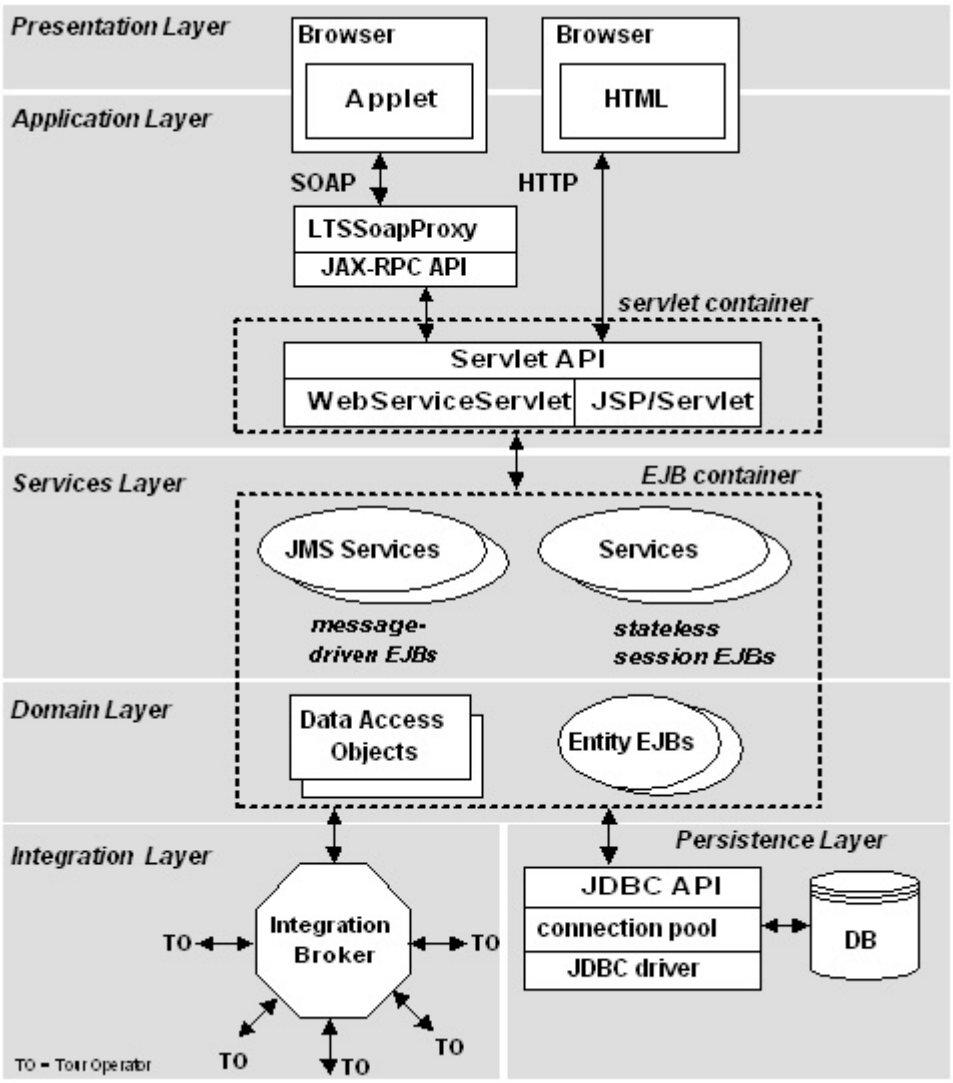

Figure 1: HolidayTrader architecture overview

The following table shows the main characteristics of the HolidayTrader system:

Number of applications	**5**
Number of session and message-driven beans	**25**
Number of entity beans	**50**
Number of Web Services	**60**
Number of Java classes	**7500**

The system had to be automatically built and deployed in five environments: development, stable development, load & performance test, pre-production, and production. The environments differed in hardware and operating systems.

Build & Deployment process overview

Having the above in mind it is not surprising that our build and deployment process was not simple. In fact, it was a full-time job for one person to maintain this process and adapt it continually to changing needs and requirements of all involved teams.

In detail, the build and deployment process had to fulfill the following requirements

- It had to be configurable with respect to versions of WLS, JDK, 3rd party libraries and locations of components, tools, and databases.
- It had to be executable in various environments that differed in hardware platform, operating system (Windows, Solaris, Linux, etc.) and WLS configuration (e.g. single instance or clustered).
- And finally, it had to be easy to use for all teams during the whole software life-cycle from development to production.

The overall goal was to provide a well-defined procedure for setting up the system in any required environment.

Figure 2 gives an overview of this procedure:

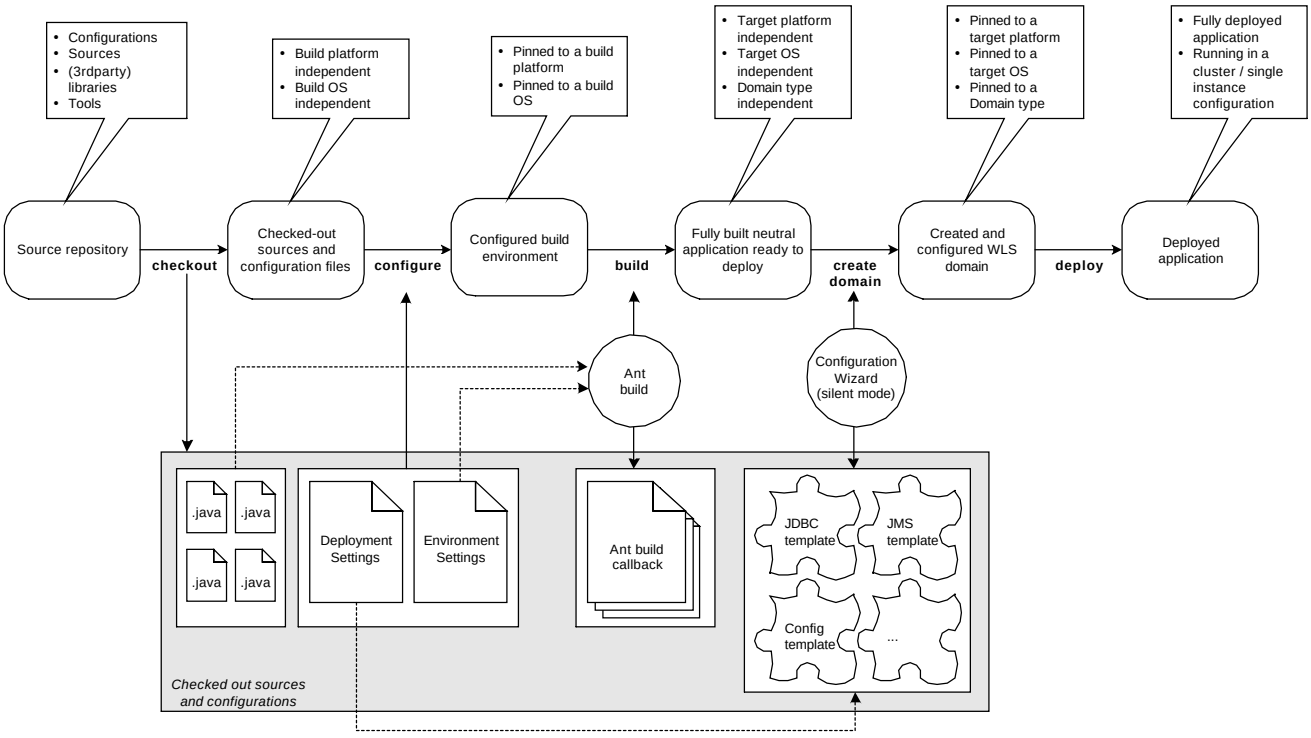

Figure 2: Build and deployment process under WLS 8.1

The build and deployment process supported all stages of setting up the system. The first step was to check out all files (source code, configurations, tools, etc.) belonging to a given release from our source control repository. This could be done in any environment we supported. The only requirement was that WLS 8.1 had to be installed on the machine used for process execution. The second step was to run a batch file with ant commands that built a “neutral” (i.e. environment independent) instance of the HolidayTrader. After that a deployment environment had to be selected from a list of supported environments. These environments were pre-configured in advance by setting parameters such as URLs, port numbers or WLS domain properties (single/clustered). Based on the selected configuration a new WLS domain

was created using the WebLogic configuration wizard in silent mode (see [1] for details). Finally the application was deployed into the newly created domain.

During the migration to WLS 9.0 the most effort was spent to upgrade the build & development process. There were only a few changes required to the application code (see below).

Migration Steps

Because of complexity of the HolidayTrader system we decided to carry out the migration in stages. The first step was to upgrade the application environment using the WebLogic *Upgrade Wizard*. In the next step we adapted our build & deployment process so far that we could manually build and run our application under WLS 9.0. In the last step we automated the build & deployment process and added support for clustered WLS instances.

The migration to WLS 9.0 was done in the development environment first. Then we used WLS 9.0 while developing a new release of the HolidayTrader. After completing the development and executing procedures for quality assurance we promoted the upgraded environment to production.

Step 1 – upgrade existing WLS 8.1 domain using *Upgrade Wizard*

In this step we upgraded the existing WLS 8.1 domain using the WebLogic *Upgrade Wizard* that is part of the WLS 9.0 delivery. This was done in our development environment containing one (non-clustered) domain.

A domain is the basic administration unit for WebLogic Server instances. It consists of one or more WebLogic Server instances that can be managed with a single administration server.

We completed the following tasks

- We checked out a stable revision of the application from our source control system and ran the build & deployment procedure in the WLS 8.1 environment.
- We installed WLS 9.0 in parallel to WLS 8.1.
- Then we copied our WLS 8.1 domain to the WLS 9.0 “domains” directory.
- Finally, we upgraded the environment using the WebLogic *Upgrade Wizard* following instructions as described in [2].

Issues & Solutions

Although the *Upgrade Wizard* did not report any error, we could not start up the WebLogic server in the upgraded environment. In fact, some manual adaptations were still required

- The xom library (xom-1.0.jar) had to be upgraded because of differences between JDK 1.4.x and 1.5.x.
- A script file for setting environment variables and paths had to be adapted manually.
- The “pointbase” section in the migrated “startWeblogic.cmd” file had to be removed because of conflicts with a Hypersonic database we used. We replaced this section with commands for setting up our own environment variables and paths.
- All occurrences of the former “./applications” path in the migrated “/config/config.xml” file had to be replaced with the new “./autodeploy ” path. This replacement had to be done in all configuration files of the deployed applications.

As soon as all the above changes were done we could start up our system and run it under WLS 9.0. We conducted nearly all jUnit tests to make sure that everything worked fine.

However the upgraded environment was still not ready to be used for development.

Step 2 – build and deploy the application under WLS 9.0

The next step was to get our build & deployment process rough-and-ready under WLS 9.0. The goal was to identify and resolve issues in the process as a preparation for final upgrade in Step 3.

- We began with redefining the environmental variables WL_HOME and JAVA_HOME and other references to WLS 9.0 and JDK 1.5 installations paths in various property files used by the build process.
- After that we executed the build process trying to resolve one problem after another. Most of these issues were related to the new JDK or caused by missing libraries (see below). Because of complexity of our build process this was a rather time consuming procedure.
- Finally, we managed to build, deploy and start up our system in the upgraded environment.

However, the build & deployment process ran in the development environment only and was not completely automated yet.

Issues & Solutions

- As mentioned before, our build & deployment process was designed to be executed in any environment we supported, even if there was no pre-created WLS domain. In this case a domain was automatically created using a scripting interface of the WebLogic configuration wizard. This interface was deprecated in WLS 9.0. Consequently, we created domains manually and postponed solving this problem to Step 3 (final migration).
- Our jar-signing procedure used “sunrsasign.jar” library that was no longer shipped with JDK 1.5. Thus we copied this jar-file from the previous installation to the folder <BEA_HOME>\jdk150_03\jre\lib.
- Because we used the XML-Bean API we had to add the path to “xbean.jar” to CLASSPATH.
- Our build process did not require any WLS installation on the machine it was executed on (WLS installation was necessary for executing the deployment process only). Of course, we had to provide some WLS libraries in order to resolve include directives in the source code. In the WLS 8.1 environment we needed the following jars
  - weblogic.jar
  - webserviceclient+ssl.jar
  - webservices.jar

  It turned out that under WLS 9.0 a few additional files were required
  - weblogic.policy
  - xbean.jar
  - persistence/persistence.install
  - schema/weblogic-container-binding.jar

Step 3 – Final migration

In this step we focused on the following tasks

- Automating configurations and build & deployment processes in all environments.
- Resolving issues related to JDK 1.5.

Automating configurations and build & deployment processes

Since our environments differed in hardware and operating systems we had to use environment dependent configurations. We provided a specific set of configuration parameters for each environment so that in a given environment the whole build & deployment could be run automatically without any manual adaptations.

However, as mentioned above, we discovered the problem with automatic domain generation. Our scripts had to be changed because the configuration wizard interface in silent mode was deprecated in WLS 9.0. Under WLS 8.1 we used this interface to create a new domain and then to add the JMS and JDBC configurations.

A solution was to use the new WebLogic Scripting Tool (WLST) instead of the configuration wizard interface (see [3] for details).

Another problem was caused by the fact that the structure of the config.xml file changed in WLS 9.0. Under WLS 8.1 we used environment specific templates for generating config.xml files. Consequently, we had to adapt these templates and we decided to rework the whole generation process completely. It was a good opportunity to use WLST consistently for all management tasks.

Figure 3 summarizes the above considerations.

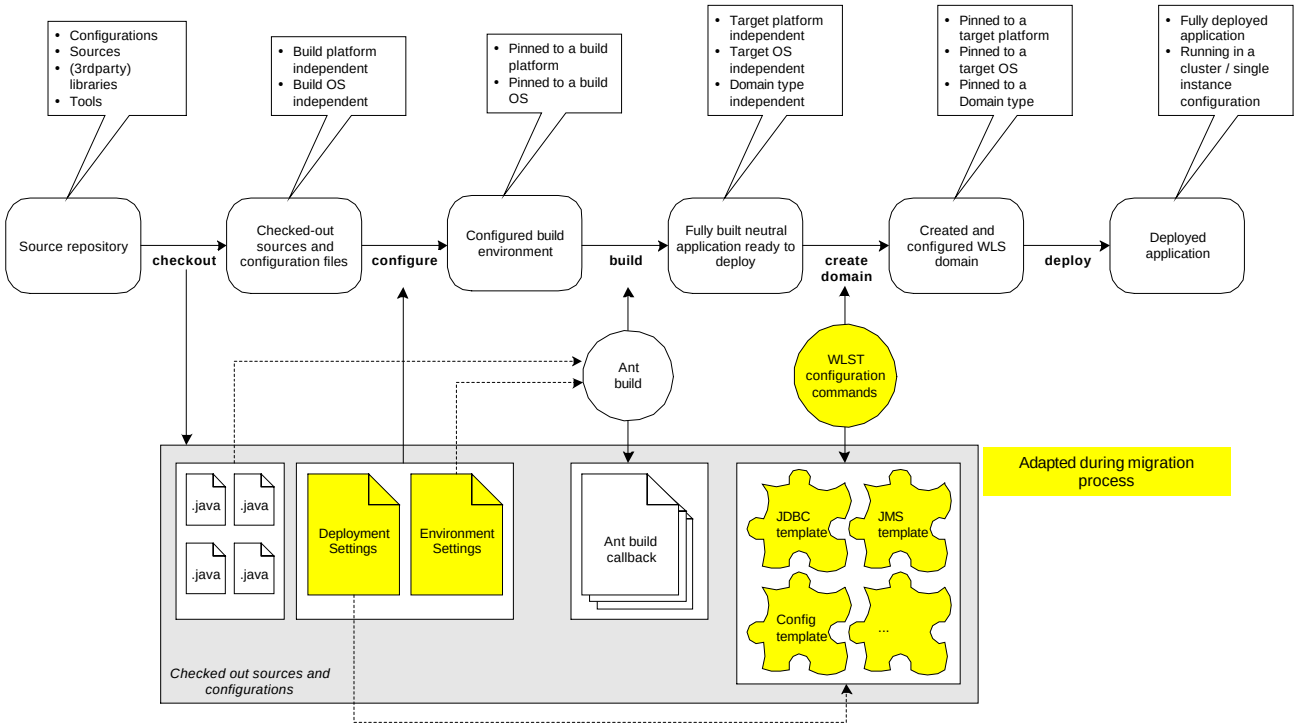

Figure 3: Changes to the build & deployment process

Resolving issues related to JDK 1.5

As soon as the above migration steps were completed, we could run extensive tests. Load & performance tests revealed however that under certain circumstances the SOAP exception handling did not work properly. We examined the problem and found that in JDK 1.5 the buffer length for SoapFaultExceptions was limited to ca. 4000 Bytes. As a result, we had to rework the handling of SoapFaultExceptions to solve the problem.

The last issue was due to special requirements concerning our roll-out process. Client components had to support both JDK 1.4.2 and JDK 1.5. This requirement resulted in a complex build process because some components had to be compiled twice using different JDKs.

Total Migration Effort

The total effort required for the migration of the HolidayTrader as described above was less than 3 person-months. The most time-consuming part was Step 3 (approx. 75% of the whole effort).

Next Steps

After migrating the HolidayTrader as-it-was (i.e. making only the really necessary changes) we considered a redesign of the system in order to take advantage of some new features of WLS 9.0. Migration is a good opportunity to carry out such a redesign.

In our case we decided to replace a proprietary timer service by the standard EJB Timer Service (see e.g. [6] for details).

Prior to EJB 2.1, there was no standard way of implementing timer services for scheduling tasks running within the EJB container. As a workaround we implemented a job scheduling using JMS API.

Fortunately WLS 9.0 added support for EJB Timer Service in compliance with EJB 2.1. This service enables one to schedule a notification at a particular time, at the end of an elapsed period of time, or at recurring intervals.

We discovered however an issue with the Timer Service in WLS 9.0. A WebLogic server instance did not start up properly in development mode when it contained an active timer service. A workaround was to configure all servers in a domain to run in production mode.

Summary

This article provides a number of best practices for moving your applications to WebLogic Server 9.0. It describes some crucial tasks that architects and developers should be aware of while migrating to WLS 9.0.

During the migration process you need to move not only the application to the target environment, but also third-party software, configurations and the build & deployment process.

References

[1] Streamline Configuration of WebLogic 8.1 Projects: http://www.ftponline.com/weblogicpro/2004_07/magazine/features/ogourment/default_pf.aspx

[2] Upgrading WebLogic Application Environments: http://e-docs.bea.com/common/docs90/upgrade/index.html

[3] Creating and Configuring WebLogic Domains Using WLST Offline:
http://e-docs.bea.com/wls/docs90/config_scripting/domains.html

[4] New Features and Enhancements J2SE 5.0
http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html

[5] What's New in WebLogic Server 9.0:
http://e-docs.bea.com/wls/docs90/notes/new.html

[6] Generic Timer using EJB Command pattern:
http://www.theserverside.com/patterns/thread.tss?thread_id=30093

[7] http://wldj.sys-con.com/read/45563.htm

About the authors:

Thomas Kruse has been working in the IT industry for 16 years. He has experience with J2EE technology since five years and has completed two large scale J2EE applications during this time. His professional background ranges from developer over system designer to system architect.

URL: www.kruse-it.de

e-mail: thomas.kruse@kruse-it.de

Alois Lechicki, PhD, is a principal consultant for Softlab, Munich (Germany). He has extensive experience in software engineering and technical architecture. He consults as an architect and guides development teams building solutions that help improve business partner integration, application integration, and process automation.

URL: www.softlab.de

e-mail: alois.lechicki@softlab.de