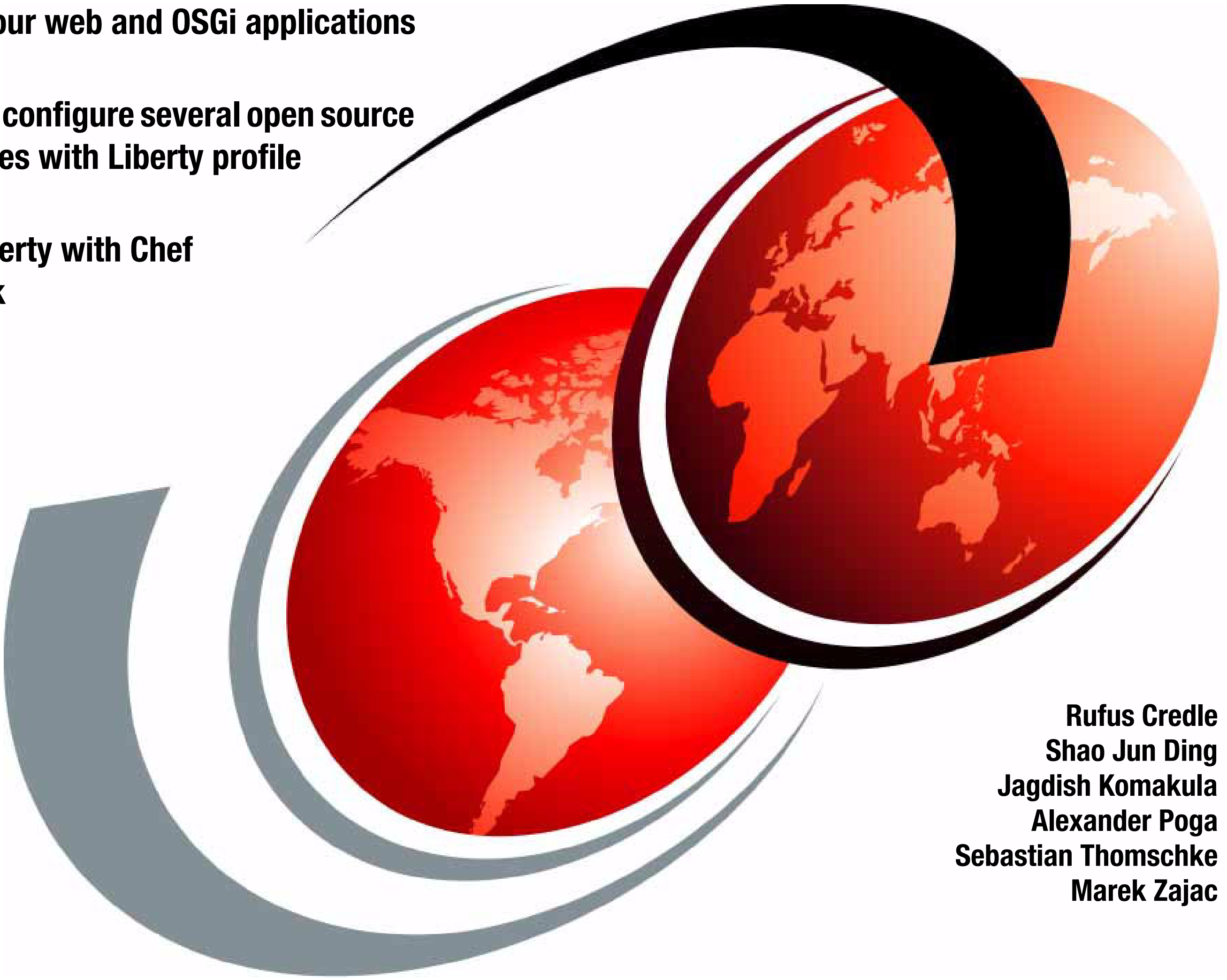
IBM® WebSphere®

Configuring and Deploying Open Source with WebSphere Application Server Liberty Profile

Learn about Liberty profile to develop and and test your web and OSGi applications

Install and configure several open source technologies with Liberty profile

Deploy Liberty with Chef framework



Rufus Credle
Shao Jun Ding
Jagdish Komakula
Alexander Poga
Sebastian Thomschke
Marek Zajac

Redbooks

ibm.com/redbooks

International Technical Support Organization

Configuring and Deploying Open Source with WebSphere Application Server Liberty Profile

March 2014

SG24-8194-00

Note: Before using this information and the product it supports, read the information in “Notices” on page v.

First Edition (March 2014)

This edition applies to IBM WebSphere® Application Server V8.5.

Contents

Notices

Trademarks

Preface

This IBM® Redbooks® publication explains the capabilities of IBM WebSphere® Application Server Liberty profile, which is extremely lightweight, easy to install, and fast to use. Liberty profile provides a convenient and capable platform for developing and testing your web and OSGi applications. The Liberty profile server is built using OSGi technology and concepts. The fit-for-purpose nature of the run time relies on the dynamic behavior inherent in the OSGi framework and service registry. As bundles are installed or uninstalled from the framework, their services are automatically added or removed from the service registry. The result is a dynamic, composable run time that can be provisioned with only what your application requires and responds dynamically to configuration changes as your application evolves.

This book can help you install, customize, and configure several popular open source technologies that can be deployed effectively with the WebSphere Application Server Liberty profile.

The following popular open source toolkits for the Liberty profile server were selected for this book based on significant enhancements they provide to the web application development process:

- Apache Maven
- Spring Framework
- Hibernate
- Jenkins
- Opscode Chef
- Arquillian
- MongoDB

In this book, the *Todo* sample demonstrates the use of multiple open source frameworks or toolkits with the Liberty profile server including Maven, MongoDB, Spring, JPA, Arquillian, Wicket, Vaadin, and others. The Todo sample is a simple application that can be used to create, update, and delete todo items and todo lists, and put the todo items into a related todo list.

Authors

This book was produced by a team of specialists working in various locations around the world with the International Technical Support Organization (ITSO), Rochester Center.

Rufus Credle is a Certified Consulting IT Specialist at the ITSO, Research Triangle Park NC Center. In his role as Project Leader, he conducts residencies and develops IBM Redbooks and Redpapers™ with a team of IBM subject matter experts, IBM Business Partners, and clients around the globe on the subject of network operating systems, enterprise resource planning (ERP) solutions, voice technology, high availability, clustering solutions, web application servers, pervasive computing, IBM and OEM e-business applications, WebSphere Commerce, IBM industry technology, IBM CICS® (Customer Information Control System) family of application servers and connectors, System x®, and IBM BladeCenter®.

Rufus has held various positions during his IBM career, including assignments in administration and asset management, systems engineering, sales and marketing, and IT services. He has a BS degree in Business Management from Saint Augustine’s College. Rufus has been employed at IBM for 33 years.

Follow Rufus on Twitter:
http://twitter.com/rcredle1906

Join Rufus’ network on LinkedIn:
http://www.linkedin.com/pub/rufus-p-credle-jr/1/b/926/

Shao Jun Ding is a Software developer for the IBM WebSphere Application Server Development organization. She is based in Beijing, China Software Development Laboratory. Iris has over 10 years of experience in the IT industry and focuses primarily on Java and Java Platform, Enterprise Edition technologies.

Jagdish Komakula is a Senior Staff Software Engineer and has over nine years of IT experience in WebSphere administration, Java, Java Platform, Enterprise Edition, XML, service-oriented architecture (SOA), and related technologies. He holds a Master's degree in Information Systems and is currently part of the Worldwide Competitive Migration Team. He was a Technical Leader for SIBus Test Team and Project Leader for WebSphere Application Server Functional Verification Test Team at the IBM Software Labs, Bangalore, India. Jagdish is focused and enjoys delivering measurable business value interacting with WebSphere customers. He is an IBM Certified WebSphere 8.5 Network Deployment Administrator and also coauthored *WebSphere Application Server V7 Migration Guide*. Jagdish has presented many technical articles at various conferences such as RTLE, APQSE, and WTC.

Alexander Poga is a Software Engineer with the IBM Australian Development Lab, based in Perth, Western Australia. He works on the development team for the Fault Analyzer for z/OS® product, part of the Problem Determination Tools suite for z/OS mainframes. His areas of expertise include Java, C, and JavaScript programming for zSeries® mainframes, Eclipse plug-ins, web services, and front-end applications.

LinkedIn: http://lnkd.in/4PdGxv

Twitter: @alexpoga

Sebastian Thomschke is a Senior IT Consultant and CEO at Vegard IT GmbH (http://vegardit.com/), a Berlin-based IT consulting company and IBM Business Partner. Its main focus on the IBM WebSphere technology stack. He has over 17 years of experience in the IT industry where he successfully fulfilled roles as Application Developer, IT Architect, Trainer, and IT Consultant. For the last eight years he has been primarily engaged in WebSphere Application Server and Portal Server development and deployment projects. Sebastian has a degree in Business Administration from the Berufsakademie Berlin, Germany.

Marek Zajac is a Software Architect with the IBM Krakow Lab, which is based in Krakow, Poland. He works with the WW BP Technical Professional team and is responsible for delivering WebSphere technical support and enablement to the IBM Business Partners in Central and Eastern Europe (CEE) and Middle East Africa (MEA). He has 13 years of experience in the IT industry and, for the last seven years has focused on Java and Java Platform, Enterprise Edition. His specialities are WebSphere Application Server, IBM Business Process Management (BPM), and WebSphere Message Broker.

LinkedIn: http://www.linkedin.com/in/marekzajac

Thanks to the following people for their contributions to this project:

Tamikia Barrow
International Technical Support Organization, Rochester Center

Alasdair Nottingham, WebSphere Application Server Liberty Profile Development Lead
IBM Hursley

Anita Rass Wan, Product Manager - WebSphere Application Server
IBM Toronto

Now you can become a published author, too!

Comments welcome

Stay connected to IBM Redbooks

- Find us on Facebook:

  `http://www.facebook.com/IBMRedbooks`

- Follow us on Twitter:

  `http://twitter.com/ibmredbooks`

- Look for us on LinkedIn:

  `http://www.linkedin.com/groups?home=&gid=2130806`

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  `https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm`

- Stay current on recent Redbooks publications with RSS Feeds:

  `http://www.redbooks.ibm.com/rss.html`

1

WebSphere Application Server Liberty profile

Along with the development of the IT industry, new challenges have occurred for application servers:

- Application servers require readily deployable artifacts. It requires a test-driven development methodology and the development must be under a continuous integration mode. The application server needs to support rapid application development and deployment.
- Software should be modularized and should be more easily assembled. This leads to the requirements for composable, dynamic, and fast application server run time.
- Modern programming models, such as NoSQL database, RESTful web services and responsive UI are more popular. Application servers must be fast and open to adopt these new programming models.
- Open source is becoming more useful. Application servers must work seamless with different open source frameworks.

The WebSphere Application Server Liberty profile is extremely lightweight, easy to install, and fast to use. It therefore provides a convenient and capable platform for developing and testing your web and OSGi applications. The *Liberty profile server* (short name) is built using OSGi technology and concepts. The fit-for-purpose nature of the run time relies on the dynamic behavior inherent in the OSGi framework and service registry. As bundles are installed or uninstalled from the framework, their services are automatically added or removed from the service registry. The result is a dynamic, composable run time that can be provisioned with only what your application requires and responds dynamically to configuration changes as your application evolves.

This book uses multiple samples to demonstrate how WebSphere Application Server Liberty profile can be used to address these challenges. This chapter has an overview of WebSphere Application Server Liberty profile and its strength.

1.1 Overview of Liberty profile server

The Liberty profile server is a simple, lightweight development and application runtime environment that offers these benefits:

- Simple to configure: Configuration is read from a single XML file with text-editor syntax.
- Dynamic and flexible: The run time loads only what your application needs and constructs the run time in response to configuration changes.
- Fast: The server starts in under five seconds with a basic web application.
- Extensible: The server provides support for user and product extensions, which can make use of system programming interfaces (SPIs) to extend the run time.

1.2 Strengths of Liberty profile server

The WebSphere Application Server Liberty Core offers great advantages when used as both a development and production run time. The Liberty profile server is both lightweight and capable, particularly when considering the ability of third parties to extend and enhance the available features. Creating a configurable run time with a custom application that can be running in seconds becomes easy and time-efficient. The learning curve to understand the new product can be short because the configuration is kept in a single file and all parameters are organized in a concise way.

1.2.1 Simple configuration

The server configuration, from the user perspective, is only a single `server.xml` file that contains all needed information. This WebSphere Application Server Liberty Core V8.5.5 configuration file has many optional parameters used for specific scenarios. You can find them in the information center:

http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.core.doc%2Fautodita%2Frwlp_metatype_core.html

Users can easily configure the server manually by editing the `server.xml` file or by using the Eclipse and the WebSphere Application Server Developer Tools. It provides the graphical tool to manage the server properties and deploy applications. For the production environment, having a set of resources that allow tuning the run time properly and also doing the troubleshooting when needed is important. For instructions to tune the Liberty profile server, go to the following web page:

http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.core.doc%2Fae%2Ftwlp_tun.html

1.2.2 Runtime composition with features and services

The composable nature of the Liberty profile server is based on the concept of features. A *feature* is a unit of functionality. Features can overlap, and they can include other features.

The Liberty profile server process consists of a single JVM, the Liberty kernel, and any number of optional features. The feature code and most of the kernel code runs as OSGi bundles within an OSGi framework. Features provide the programming models and services required by applications. You can choose which optional features should be enabled according to your application requirements.

For a list of the main Liberty features, go to the following web page:

`http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/topic/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/rwlp_feat.html`

1.2.3 Developer First focus

With Liberty profile server, you can do rapid development and deployment to meet with the agile development trend. Liberty profile server is an open source software alternative with WebSphere Quality of Service.

Fast and no-cost download for developer's desktop

Liberty profile server is fast and no cost for developer desktop use. It can be downloaded and installed from Eclipse.org or WASdev.net. To download from WASdev.net, go to this address:

`http://wasdev.net/downloads`

Development tools are also available: WebSphere Application Server Developer Tools for Eclipse. The WebSphere Application Server Developer Tools product is available through the Eclipse Marketplace. You can also use IBM Rational® Application Developer for your development. For more information about WebSphere Application Server developer tools and access to the tool, see the following website:

`https://www.ibmdw.net/wasdev/`

Rapid development and deployment

You deploy an application in Liberty profile server by either dropping the application into server's `dropins` directory, or by adding an application entry to the server configuration (`server.xml`) file.

By default, the `dropins` directory is automatically monitored. If you drop an application into this directory, the application is automatically deployed on the server. Similarly, if the application is deleted from the directory, the application is automatically removed from the server. For applications that are not in the `dropins` directory, you specify the location using an application entry in the server configuration file. The location can be on the file system or at a URL.

Although there is no distinction between installing and starting an application, installed applications can be stopped and restarted. By default, the Liberty profile server monitors deployed applications for changes. Updates to static files (HTML, CSS, or JavaScript) or JSP files are detected and served immediately. Changes to servlet classes cause an automatic restarting of the application. So, in your development time, updating your application is quick and you do not need to redeploy the application. The server can monitor and restart the application, if needed, to retrieve your new changes.

When you finish development, if you want to distribute the final result to users, you can package the Liberty profile server from the command line. Then, you can store this package, distribute it to colleagues, use it to deploy the installation to a different location or to another machine, or embed the installation in a product distribution. For details about how to package and distribute your Liberty profile server and application, go to the following web page:

`http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.core.doc%2Fae%2Ftwlp_setup_package_server.html`

1.2.4 Multiple programming model support

The Liberty profile server is a certified Web Profile; you can create web applications for the Java Platform, Enterprise Edition (Java EE) Web Profile, standard on the Liberty profile server.

The Liberty profile server supports a subset of the Java EE 6 stack also, for example, web services technologies, enterprise application technologies, and more.

The Liberty profile server also supports OSGi applications in all editions. The following technologies are supported for OSGi applications (with a reference to the specification where appropriate):

- Web Application Bundles (OSGi R4.2 Enterprise, Chapter 128)
- Blueprint Container (OSGi R4.2 Enterprise, Chapter 121)
- Blueprint Transactions
- Blueprint Managed JPA
- JNDI (OSGi R4.2 Enterprise, Chapter 126)
- OSGi application of Java EE technologies that are supported by the profile

A complete list of the technologies that are supported by the Liberty profile server are at the following information center website:

http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/topic/com.ibm.websphere.wlp.nd.doc/ae/rwlp_prog_model_support.html

1.2.5 Easy extensibility for custom features and third-party components

The Liberty profile server supports direct extension of the run time using product extensions. A product extension allows custom content to be added to a Liberty installation in a way that avoids conflicts with the base content of the product and with other product extensions. A product extension is defined using a simple properties file (`<extensionName>.properties`) in the following directory:

```
${wlp.install.dir}/etc/extensions/
```

This naming convention helps to ensure that each product extension has a unique name. The unique name, in turn, is used as a prefix to avoid collisions when specifying extension-provided features and configuration in `server.xml` file.

1.2.6 Easy access

The Liberty profile server is a small download file that you can extract to install. The download size is only 50 MB and the installed size is approximately 67 MB. The installation time on the developer system is under ten seconds.

You can also first install WebSphere Application Server Developer Tools for Eclipse and then use this tool to download the WebSphere Application Server Liberty profile. This installation option is complete in only three minutes.

Liberty profile server supports a broad list of operating systems: IBM AIX®, HP, IBM i, Linux, Solaris, Windows, and z/OS. The Liberty profile server is also supported, for development on Mac OSX.

The Liberty profile server runs on Java Version 6 or later, regardless of the vendor that provides it.

1.2.7 Fast and small footprint

The Liberty profile server startup is fast. It can be started in approximately three seconds, and even in debug mode it can be started within five seconds.

The Liberty profile server memory footprint is small at less than 50 MB for TradeLite benchmark.

1.2.8 Compatibility with WebSphere Application Server full profile

The WebSphere Application Server Liberty profile shares many core technologies and capabilities with the WebSphere Application Server *full profile* server. Any application that runs on the Liberty profile server can also run on the WebSphere Application Server full profile server, although the reverse is not necessarily true. One consideration is the differences in the JSR specifications supported by each version of the server. Depending on your application, you might need to verify what WebSphere Application Server profile version you must use. Go to the following web page for support details:

`http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.nd.doc%2Fae%2Frwlp_prog_model_support.html`

Other differences also exist between profiles in terms of configuration, including parameter names and their default values. The following list summarizes the primary areas that differ between the WebSphere Application Server full profile and the Liberty profile server:

- Time values

  In the WebSphere Application Server Liberty profile, most properties are represented by units of time. In the WebSphere Application Server full profile, they are stored as units of seconds, milliseconds, or minutes depending on the setting.

- Class loading

  The main difference is that the WebSphere Application Server full profile uses and exposes many open source libraries to applications running on the server. For example, classes under the `org.apache.*` location are visible in the application. The following open source packages are available to applications on the WebSphere Application Server:

  - commonj
  - org.apache.axiom
  - org.apache.axis2
  - org.apache.bval
  - org.apache.http
  - org.apache.wink
  - org.apache.xerces
  - org.apache.commons.[beanutils | codec | collections | digester | discovery | el | fileupload | httpclient | io | jxpath | lang | lang3 | logging | pool]

  You might encounter problems when your application uses the same libraries at different versions deployed within your application. To fix this issue, usually you need to change the class loader from the parent first to the parent last. WebSphere Application Server Liberty profile exposes only the specification API and IBM APIs to applications, by default. This means you do not have to change the class loader policy. But you must remember the default action when migrating your application from the Liberty profile server to the full profile.

- Server properties

  If you are familiar with the WebSphere Application Server full profile, you already know there are many parameters, for example, data sources, web container properties, thread pools and others. Usually you configure the server using the web based admin console or wsadmin command line tool. The WebSphere Application Server Liberty profile has only one single configuration XML file that contains all server settings instead of many XML files that contain parameters in the WebSphere Application Server full profile. An important aspect to mention here is that name of those parameters and their default values might differ among the various WebSphere Application Server profiles. Be aware of this when you plan to switch from Liberty profile server to WebSphere Application Server full profile.

  For WebSphere Application Server Liberty profile, see the following web page to learn about server parameters, their descriptions, and default values.

  `http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.nd.doc%2Fautodita%2Frwlp_metatype_4ic.html`

When you are planning to switch from the WebSphere Application Server full profile to Liberty profile server, consider the following differences:

- Liberty profile server supports only EJB 3.x beans.
- JNDI lookups using the `ejblocal` name space are not supported. Instead, the ejb-ref bindings must be specified using `java:global`, `java:app`, or `java:module` name. The simple-binding-name and interface binding-name elements are ignored in the `ibm-ejb-jar-bnd.xml` file.
- Java Server Faces 2.0 in WebSphere Application Server Liberty profile uses the MyFaces implementation. If you need to use a different implementation, you must include the proper libraries in the application and do not add the JSF feature in the `server.xml` file. Notice that when you change the default implementation, operations such as resource injection no longer work.
- Security differences exist between WebSphere Application Server Liberty profile and WebSphere Application Server full profile. To learn more, go to the following web page:

  `http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.nd.doc%2Fae%2Fcwlp_sec.html`
- Web Services Security in WebSphere Application Server Liberty profile is configured within the WSDL file in applications. In WebSphere Application Server, the same can be configured by applying policy sets.
- Bean validation is not supported in the Liberty profile server when deployed in OSGi applications.
- In the WebSphere Application Server full profile, when you want to expose an EJB3.x through web services, a web archive (WAR) Router web project is generated. This is not needed in WebSphere Application Server Liberty profile, where you can directly expose EJB using `@WebService` annotation.

2

Open source frameworks and toolkits selection

This chapter introduces several popular open source technologies that can be deployed effectively with the WebSphere Application Server Liberty profile.

The following list represents a selection of popular open source toolkits for the Liberty profile server. Each product was selected based on the significant potential enhancements they provide to the web application development process.

- Apache Maven
- Spring Framework
- Hibernate
- Jenkins
- Opscode Chef
- Arquillian
- MongoDB

2.1 Apache Maven

Apache Maven is an open source software project management tool, designed to streamline the software development lifecycle. Although it is primarily focused on Java-based software projects, Maven can be configured to work with other languages, such as C#, Ruby, and Scala.

Through configuration of a Project Object Model (POM) XML configuration file, Maven is able to automate several common tasks, including these examples:

- Building project source files
- Managing dependencies on external modules and components, such as Java libraries
- Running unit tests and reporting test code coverage
- Generating documentation and project information reports, including change logs
- Managing release distribution and mailing lists

In addition, capabilities of Maven can be enhanced to meet further requirements through its extensible plug-in architecture.

Maven build lifecycles consist of a list of ordered phases. Figure 2-1 shows how build lifecycles and Maven plug-ins are organized.

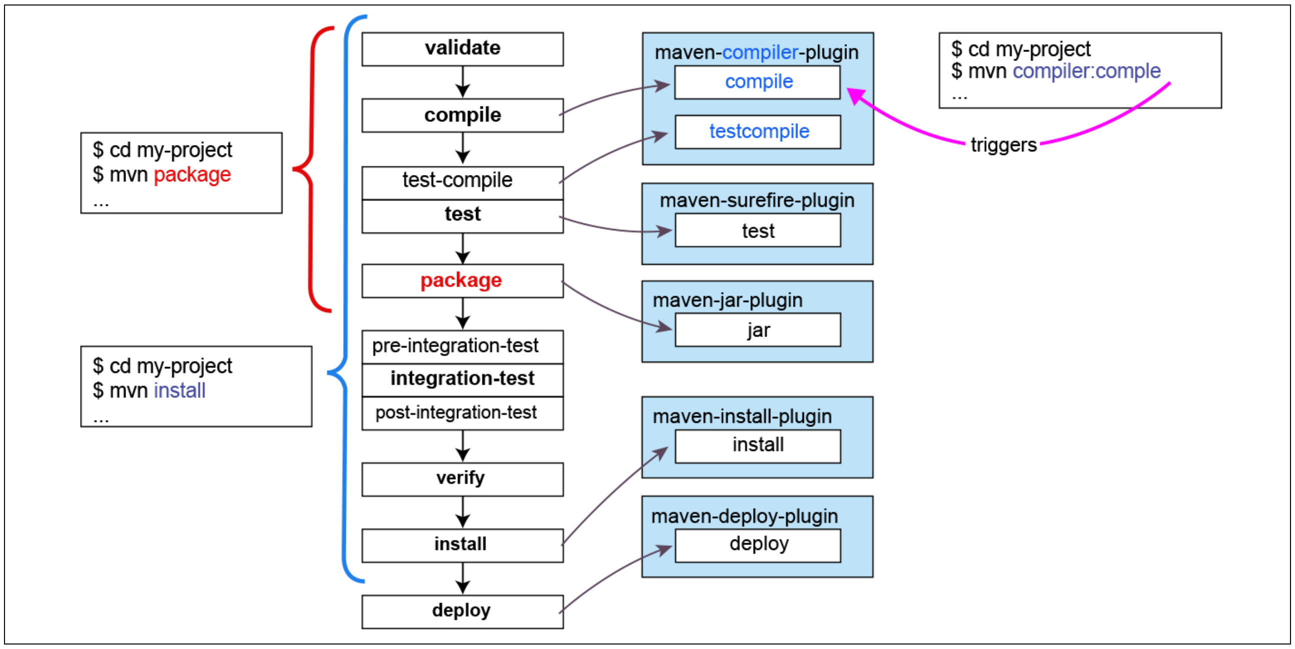


Figure 2-1 Default Maven lifecycle plug-in bindings

The Liberty profile server provides a Maven plug-in that can be used for automating Liberty profile server specific tasks. After adding the Liberty profile server Maven plug-in from the WebSphere Application Server Developer Community (WASdev) to your POM XML file, the following “goals” become available:

- liberty:create-server
- liberty:start-server
- liberty:package-server
- liberty:stop-server
- liberty:undeploy
- liberty:install-apps

2.2 Spring Framework

The Spring Framework is a modular Java development platform that provides several features to support the enterprise application development process. Spring uses a model of dependency injection or *Inversion of Control* (IoC) whereby Java components can be created and wired together by declaring relationships using XML and annotations. This enables an application developer to focus on the business logic using "plain old Java objects" (POJOs), leaving the infrastructure management to Spring.

As an example, each of the following operations can be performed using local Java methods without needing to directly interact with their respective environment APIs:

- Executing database transactions
- Performing remote procedures
- Performing management operations using the Java Management Extensions (JMX)
- Handling messages using the Java Message Service (JMS)

2.2.1 Spring Framework modules

The Spring Framework modules are organized into the categories: core container, data access and integration, web, aspect-oriented programming, and testing.

Core container

This category provides the fundamental components of the framework, using Inversion of Control (IoC) and dependency injection for configuration, context, and lifecycle management of Java objects. In addition, the Expression Language module provides various utility functions that include support for querying and manipulating objects at run time.

Data access/integration

This category facilitates integration with numerous popular data access frameworks to support JDBC, Object Relational Mapping (ORM), Object/XML Mapping (OXM), JMS, and transaction management. Supported frameworks include JDBC, Hibernate, JPA, JDO, and iBatis for ORM, and also JAXB, Castor, XMLBeans, JiBX, and XStream for OXM.

Web

This category includes several useful web components and utilities, including request parameter parsing, multi-part request handling (as used in uploading files) the remote access framework, and a model view controller (MVC) framework for web applications.

Aspect-oriented programming and instrumentation

Aspect-oriented programming (AOP) is a technique to alleviate issues with cross-cutting concerns by encapsulating behaviors between multiple classes into reusable modules. Cross-cutting concerns are parts of a program that rely on or affect many other components in a software system, which cannot be cleanly implemented in object oriented or procedural programming. An example might be an application-wide logging or caching system that requires code duplication or complicated dependency linkages between components.

Testing

This category includes modules to support the testing of Spring components by providing component mocking utilities and also consistent loading and caching of Spring ApplicationContexts.

2.3 Hibernate

Hibernate is an Object Relational Mapping (ORM) library for Java applications. Using a lightweight persistence framework, Hibernate handles the mapping of Java objects to tables in a traditional relational database.

To provide this functionality, the Hibernate data query and retrieval facilities abstract away from the selected database SQL calls and result set handling. This allows the developer to focus on the features of an application, without having to worry about interacting with a database when the developer needs to store or find objects.

Hibernate can use Java annotations or XML mapping documents to describe the following mappings:

- Java objects to relational database tables
- One-to-many and many-to-many relationships between objects
- Reflexive one-to-many relationships between an object and other instances of its own type
- Java object types to SQL types (when overriding the default mappings)
- Java Enum types to columns
- Single properties to multiple columns

2.4 Jenkins

Jenkins is an open source continuous integration platform that runs under a Java web servlet container (such as the Liberty profile server).

Continuous integration is the practice of frequently building and testing software projects during development. The aim of this process is to discover defects and regressions early by automating the process of running unit and integration tests. These automated build and test cycles typically happen on a regular schedule (such as every night) or even after each change is delivered.

Jenkins is able to integrate with a large variety of frameworks and toolkits, using its extensive library of available plug-ins, including the following items:

- Source code management and version control platforms including CVS, Subversion, Git, Mercurial, Perforce, Clearcase, and RTC
- Build automation tools such as Apache Ant and Maven, and also standard operating system batch and script files
- Testing frameworks such as JUnit and TestNG
- RSS, email, and instant messenger clients for reporting results in real time
- Artifact uploaders and deployers for several integration platforms

2.5 Opscode Chef

Opscode Chef is an application and server configuration framework that is designed to automate the process of provisioning and deploying resources. Chef provides a way to model infrastructure and processes in code so that they become testable, versioned, and repeatable.

The *chef-client* uses abstract reusable definitions of system configurations named *cookbooks* and *recipes*. These configuration files describe how each part of the server infrastructure should be built and managed, whether it is in a physical, internal, or cloud-based server environment. When new hardware is added, the chef-client needs to know only which cookbooks and recipes to apply. For example, a cookbook might define everything that is required to install, configure, and manage the Liberty profile server on a node.

Through integration with frameworks such as Jenkins, Chef also enables a model of *continuous delivery*, where updates can automatically be deployed into production after completing the continuous integration testing and approval process.

In a traditional client/server configuration, the Chef framework consists of three main components:

- A single Chef server
- A chef-client installed on each node (a physical, virtual, or cloud server)
- One or more workstations to be used for administration and development

In addition, Chef can also be run using a limited *chef-solo* configuration that runs without a server. This setup does, however, require that a cookbook and any dependencies are available on the same physical disk as the node.

2.6 Arquillian

Integration-testing a modern Java web application can be difficult and time consuming, because of the necessity of re-creating a realistic web container environment. Managing a simulated environment is usually achieved using mocking or extensive environment initialization and teardown between tests. Arquillian is a testing framework that aims to simplify this process by automatically handling container management, deployment and framework initialization. With this design, developers can focus on the testing of business logic with what Arquillian describes as “real tests.”

By handling the environment configuration for the tester, Arquillian allows test cases to be run against several various platforms. This is particularly helpful when various web containers are used between development and production (such as using the Liberty profile server for development of a WebSphere Application Server full profile application).

Using special Java annotations to integrate seamlessly with familiar testing frameworks (such as JUnit and TestNG), Arquillian tests can be run from a developer’s IDE or build system without any extra plug-ins, by using tools such as Ant and Maven.

2.7 MongoDB

MongoDB is an open-source, cross-platform, document-oriented database system. Based on the concept of a dynamic NoSQL structure, MongoDB differs from a traditional relational database in its ability to store documents in a format similar to JavaScript Object Notation (JSON). This data format allows for greater flexibility on the types of information that are stored (including regular files using GridFS), by eschewing the limitations of a strict schema. Example 2-1 shows two samples of documents that have different attribute structure.

Example 2-1 Two example MongoDB documents with differing attribute structure

```
{
   firstName: “John”,
   secondName: “Smith”,
   age: “20”
}

{
   firstName: “Mary”,
   secondName: “Smith”,
   hobby: “painting”,
   favouriteColor: “blue”
}
```

To support efficient query resolution, MongoDB uses an indexing system for each document. Any field can be selected for use as the document index and secondary indices are also supported.

MongoDB offers several features to support large workloads and storage requirements. Using a *replica set*, groups of MongoDB daemon processes can maintain the same data set in multiple physical locations. Data can be mirrored by establishing a “master-slave” relationship between environments where the slave maintains a copy of the master database for reading and backup purposes. This configuration has the advantage that a new master source can be selected if the current master becomes unavailable. In addition, MongoDB also supports sharding, where one large set of data is split into ranges that are distributed across multiple servers.

3

Implementing and testing backend services on Liberty profile server

In this book, the *Todo* sample is used to demonstrate the use of multiple open source frameworks or toolkits with Liberty profile server including Maven, MongoDB, Spring, JPA, Arquillian, Wicket, Vaadin, and others. The Todo sample is a simple application that can be used to create, update, and delete to-do items and to-do lists, and put the to-do items into related to-do list. It also provides a query function such as get todo items in one specific to-do list, get all to-do lists, and so on. Later, in the todo-service-api project, you can find all available APIs.

This chapter guides you from setting up the development environment through the integration testing using Arquillian. It mainly focuses on back-end service implementation and how to do an integration test with Liberty profile server automatically. See Chapter 5, "Front-end development on the Liberty profile server" on page 45 for an extended set of sample applications that incorporate graphical user interfaces.

3.1 Setting up the development environment

Preparing the development environment for our sample application requires the following steps:

1. Install JDK 7 and update your system JAVA_HOME environment variable to reference the installation directory.
2. Download and install Eclipse and the WebSphere Application Server Developer Tools from the following location:

   `https://www.ibmdw.net/wasdev/docs/developing-applications-wdt-liberty-profile/`
3. Install Apache Maven for command-line console-based builds from the following location:

   `http://maven.apache.org/download.cgi#Installation_Instructions`
4. Install the m2eclipse plug-in into Eclipse.

   `http://www.eclipse.org/m2e/`
5. Install Subclipse 1.8.x (NOT the 1.10.x) plug-in:

   `http://subclipse.tigris.org/servlets/ProjectProcess?pageID=p4wYuA`
6. Get all Todo sample-related source files from this book's FTP site and save them into your local directory.
7. Open Eclipse and go to **Window** → **preferences** → **Java** → **Install JRES**, and then select **JDK 7 home**.

8. In Eclipse, click **File** → **Import** → **Maven** → **Existing Maven Projects**, as in Figure 3-1. Click **Next**.

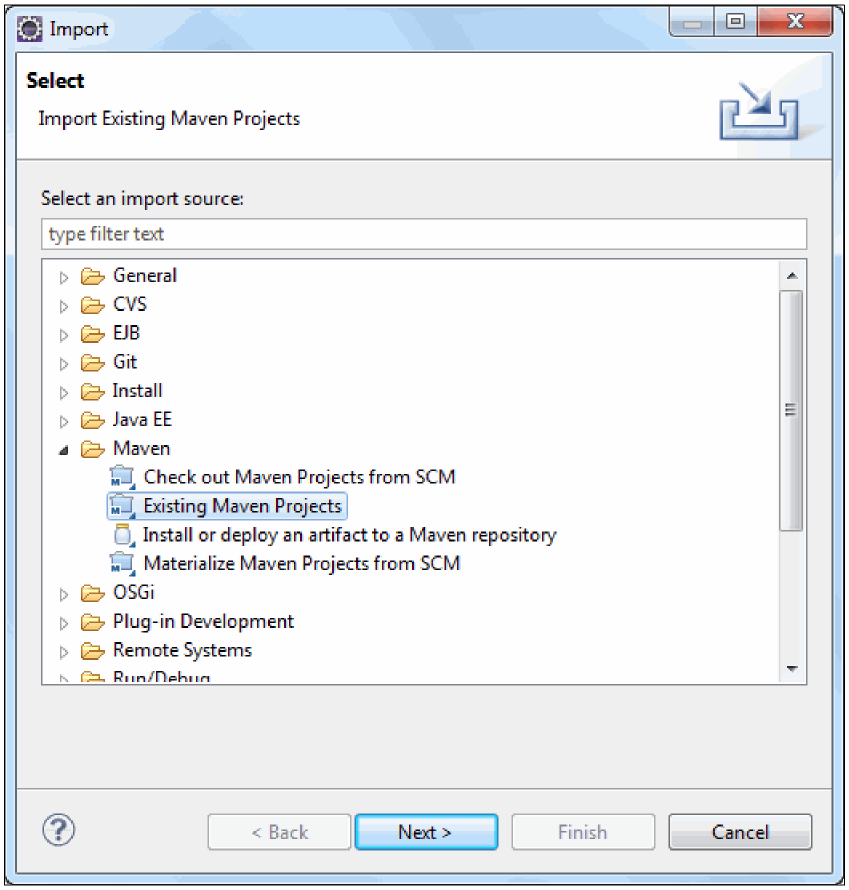
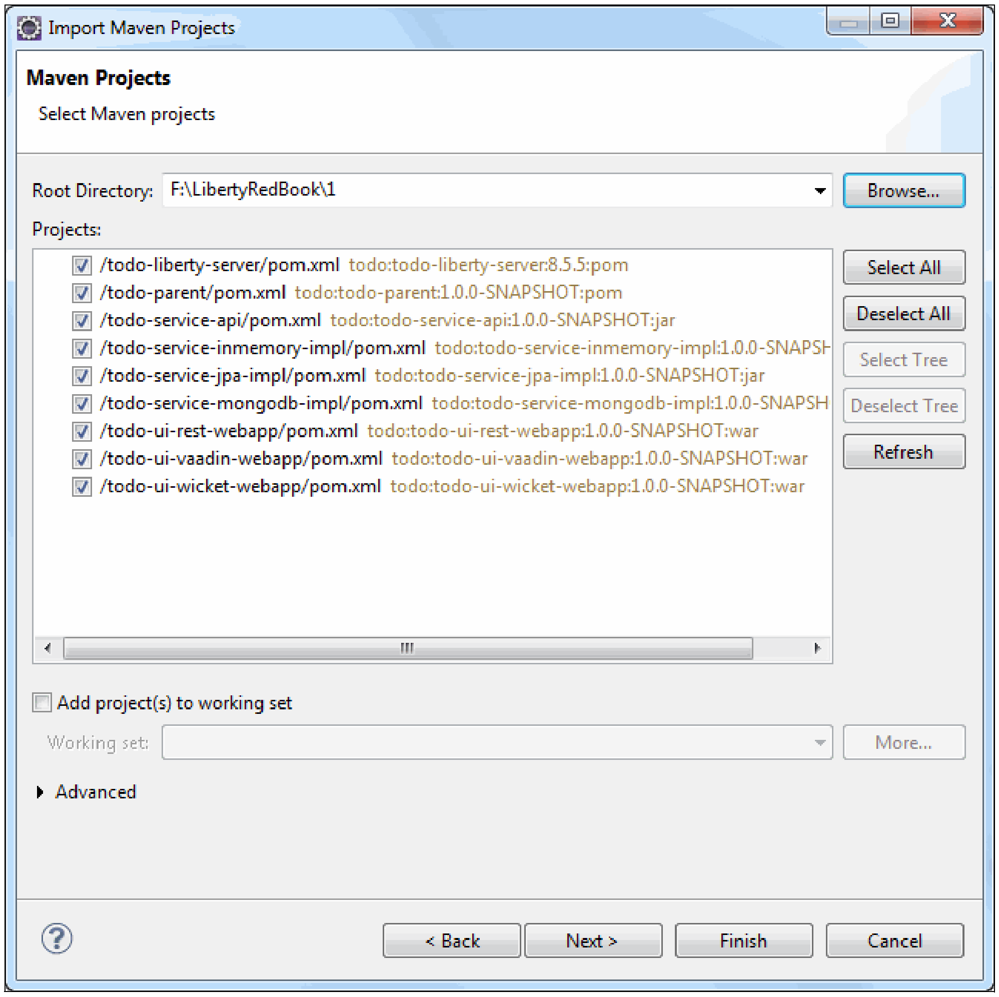


Figure 3-1 Import Existing Maven Projects

9. Open the directory where you saved the Todo sample. Ensure that all projects are selected and click **Finish**, as shown in Figure 3-2.



Root Directory: F:\LibertyRedBook\1

Projects:

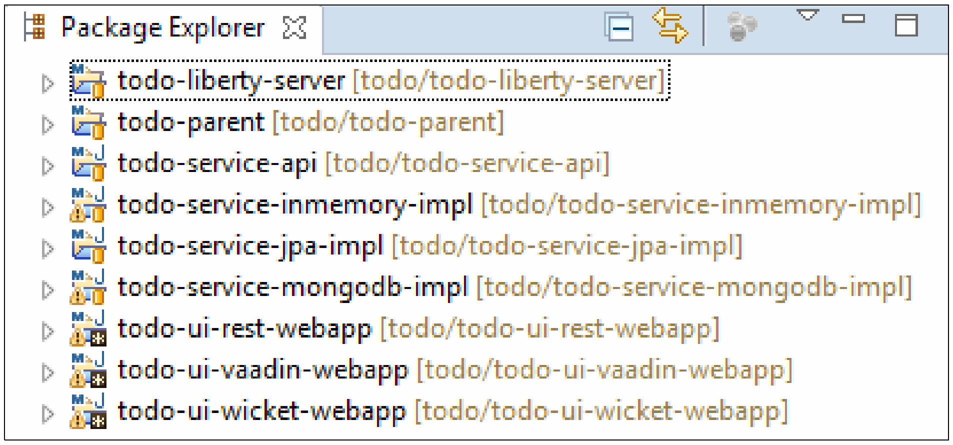
```
☑ /todo-liberty-server/pom.xml  todo:todo-liberty-server:8.5.5:pom
☑ /todo-parent/pom.xml  todo:todo-parent:1.0.0-SNAPSHOT:pom
☑ /todo-service-api/pom.xml  todo:todo-service-api:1.0.0-SNAPSHOT:jar
☑ /todo-service-inmemory-impl/pom.xml  todo:todo-service-inmemory-impl:1.0.0-SNAPSH
☑ /todo-service-jpa-impl/pom.xml  todo:todo-service-jpa-impl:1.0.0-SNAPSHOT:jar
☑ /todo-service-mongodb-impl/pom.xml  todo:todo-service-mongodb-impl:1.0.0-SNAPSH
☑ /todo-ui-rest-webapp/pom.xml  todo:todo-ui-rest-webapp:1.0.0-SNAPSHOT:war
☑ /todo-ui-vaadin-webapp/pom.xml  todo:todo-ui-vaadin-webapp:1.0.0-SNAPSHOT:war
☑ /todo-ui-wicket-webapp/pom.xml  todo:todo-ui-wicket-webapp:1.0.0-SNAPSHOT:war
```

☐ Add project(s) to working set

Working set:

Figure 3-2 Select all projects to import

Wait a few seconds. Your projects are then listed, without errors, in the Project Explorer (Figure 3-3); m2eclipse has already taken care of cross-references between projects and configured the projects accordingly.



```
todo-liberty-server [todo/todo-liberty-server]
todo-parent [todo/todo-parent]
todo-service-api [todo/todo-service-api]
todo-service-inmemory-impl [todo/todo-service-inmemory-impl]
todo-service-jpa-impl [todo/todo-service-jpa-impl]
todo-service-mongodb-impl [todo/todo-service-mongodb-impl]
todo-ui-rest-webapp [todo/todo-ui-rest-webapp]
todo-ui-vaadin-webapp [todo/todo-ui-vaadin-webapp]
todo-ui-wicket-webapp [todo/todo-ui-wicket-webapp]
```

Figure 3-3 Projects listed after they are imported into Eclipse

3.2 Project outline of the todo list sample application

Table 3-1 explains the structure of the sample application.

Table 3-1 The todo sample application project structure

Project name	Basic description
todo-parent	Holds shared Maven configuration for all projects
todo-service-api	Contains the model and service API
todo-liberty-server	Contains Liberty profile server binary
todo-service-inmemory-impl	Contains an example in-memory implementation of the todo-service
todo-service-jpa-impl	Contains an example JPA implementation of the todo- service
todo-service-mongodb-impl	Contains an example mongodb implementation of the todo-service
todo-ui-rest-webapp	Contains an example RESTful web service API and Dojo Toolkit based front-end
todo-ui-wicket-webapp	Contains an example Wicket front-end UI of the todo-service
todo-ui-vaadin-webapp	Contains an example Vaadin front-end UI of the todo-service

The relationship between each project is shown in Figure 3-4.

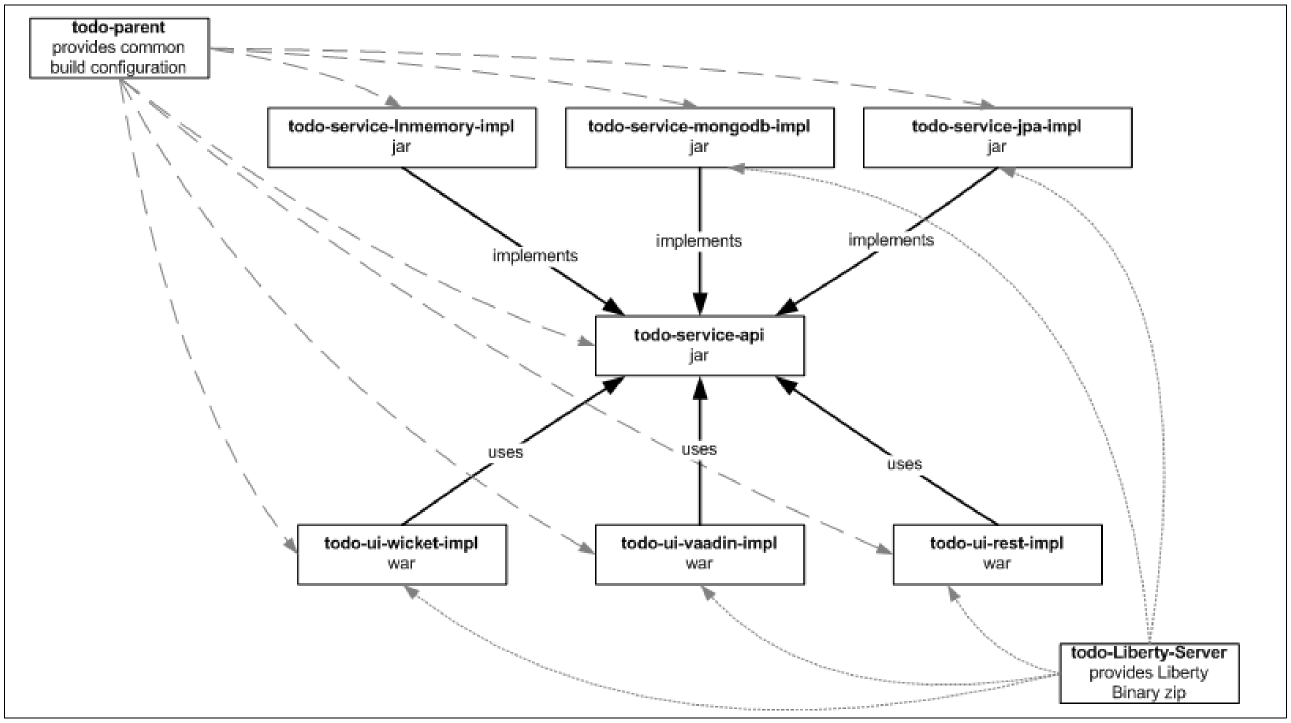


Figure 3-4 The todo sample projects relationship

3.2.1 The simple todo-parent project

The sample todo list application configuration begins in the Maven *todo-parent* parent component. Maven supports configuration inheritance, which means a Maven project can define another project as its parent project by using the `<parent>` XML node and inherits its configuration. Dependencies declared in a parent project are automatically declared as dependencies in child projects. Using the `<dependencyManagement>` and `<pluginManagement>` sections in a parent project, preconfigurations of artifact versions and plug-in configurations can be made.

This means that in all child projects, we can use the Liberty `assemblyArtifact` in the local Maven repository to create a Liberty profile server, as shown in the `pom.xml` sample file (Figure 3-5).

```
<pluginManagement>
   <!-- plugin configurations listed here apply to all child modules which 
explicitly list these plugins in there <build><plugins>...</plugins></build> 
section -->
   <plugins>
      <plugin>
         <groupId>com.ibm.websphere.wlp.maven.plugins</groupId>
         <artifactId>liberty-maven-plugin</artifactId>
         <version>1.1</version>
         <configuration>
            <assemblyArtifact>
               <!-- instruct the liberty maven plugin to use the liberty server 
binaries provided by this artifact -->
               <groupId>todo</groupId>
               <artifactId>todo-liberty-server</artifactId>
               <version>8.5.5</version>
               <type>zip</type>
            </assemblyArtifact>
         </configuration>
      </plugin>
   </plugins>
</pluginManagement>
```

Figure 3-5 Liberty Maven plug-in defination in the pom.xml file

This simplifies the configuration process by providing a single location for specifying settings such as the Java compiler version and any common dependencies. These settings however become effective only when that same dependency or plug-in is directly declared in the `pom.xml` file of the child project.

Configurations defined directly in a `pom.xml` file override similar settings defined in parent projects. If child projects are registered as modules in a parent project, then they are built as part of the parent project's build process. Executing the *mvn package* on a parent project will also trigger execution of this lifecycle phase in all declared modules. Child projects that are not registered as modules are independent from the parent project's build process and thus must be built separately.

3.2.2 The todo-service-api project

The *todo-service-api* project sits in the middle of our back-end implementation and front-end UI. All available APIs for users are defined in this project. In this way, switching the back-end implementation and front-end UI, based on our requirements, is easy.

Also, `AbstractTodoListServiceTest.java` is defined to perform functional tests against the TodoListService interface. In the MongoDB and JPA back-end implementation, `AbstractTodoListServiceTest.java` does not need to write its functional tests, but extend the abstract test class and ensure that the specific implementation complies with the test case. To make this happen, the AbstractTodoListServiceTest is packaged into the `test.jar` file (`test.jar` is a dependency) so that it can be available for all the back-end service implementation projects: in-memory, MongoDB, and JPA because, *by default*, test sources are not packaged. Figure 3-6 is just a snippet of this part in the `pom.xml` file.

```
<build>
   <plugins>
      <plugin>
         <!-- create a jar containing the test cases, so the testcases can be
referenced and reused in other projects -->
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-jar-plugin</artifactId>
         <version>2.2</version>
         <executions>
            <execution>
               <phase>package</phase>
               <goals>
                  <goal>test-jar</goal>
               </goals>
            </execution>
         </executions>
      </plugin>
   </plugins>
</build>
```

Figure 3-6 Test resources exposure in pom.xml

3.2.3 The todo-liberty-server project

The Liberty 8.5.5 binary is included in this project. This project installs the Liberty profile server assemblyArtifact in your local Maven repository. In other projects, you can use the Liberty Maven plug-in to create server that is based on this assemblyArtifact in a local Maven repository as described in the `pom.xml` file in the todo-parent project.

You can also run the **`mvn -Pregenerate-server-xsd generate-souces`** command to generate the Liberty profile server configurations schema. The generated schema is installed into the local Maven repository. Therefore, in other projects, you can reference this schema file to facilitate your editing of `server.xml` file. The snippet for this part is in the `pom.xml` file (Figure 3-7 on page 20).

```
<profiles>
   <profile>
      <!-- activate this profile to regenerate the server.xsd for the liberty 
server -->
      <id>regenerate-server-xsd</id>
      <build>
         <plugins>
            <plugin>
               <groupId>org.codehaus.mojo</groupId>
               <artifactId>exec-maven-plugin</artifactId>
               <version>1.2.1</version>
               <executions>
                  <execution>
                     <phase>generate-sources</phase>
                     <goals>
                        <goal>exec</goal>
                     </goals>
                  </execution>
               </executions>
               <configuration>
               <!-- 
http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/topic/com.ibm.websphere.wlp.core
.doc/ae/rwlp_schema_gen.html -->
                  <executable>java</executable>
                  <arguments>
                     <argument>-jar</argument>
                     <argument>
                        src/main/resources/wlp/bin/tools/ws-schemagen.jar
                     </argument>
                     <argument>src/main/resources/server.xsd</argument>
                     <argument>--locale=en_US</argument>
                  </arguments>
               </configuration>
            </plugin>
         </plugins>
      </build>
   </profile>
</profiles>
```

Figure 3-7 Liberty profile server configuration schema generation in the pom.xml file

3.2.4 The todo-service-inmemory-impl project

The *todo-service-inmemory-impl* project is a simple in-memory implementation for TodoListService. Orika is used in this project to recursively copy data from the service object to in-memory entity objects and vice versa. It was added as a dependency in the `pom.xml` file in this project. Orika is a simpler, better, and faster Java bean mapping framework and is available in the Maven central repository. Details of Orika are at the following website:

https://code.google.com/p/orika/

Because todo-service-inmemory-impl implements the TodoListService in todo-service-api, the todo-service-api project has been added as a dependency for it. Also as described in 3.2.2, “The todo-service-api project” on page 19, test-jar is another dependency.

Figure 3-8 is the dependency snippet for this project in the `pom.xml` file.

```
<dependencies>
   <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>todo-service-api</artifactId>
      <version>${project.version}</version>
   </dependency>
   <dependency>
   <!--
      high performance object mapping framework.
      used to convert service objects to entity objects and vice versa
   -->
      <groupId>ma.glasnost.orika</groupId>
      <artifactId>orika-core</artifactId>
      <version>1.4.0</version>
   </dependency>

   <!-- TEST DEPENDENCIES -->
   <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>todo-service-api</artifactId>
      <version>${project.version}</version>
      <type>test-jar</type>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Figure 3-8 Dependency definition in pom.xml

One test, `InMemoryTodoListServiceTest`, is defined. The test extends the class AbstractTodoListServiceTestm which is part of the todo-service-api project and performs functional tests against the TodoListService interface. The InMemoryTodoListServiceTest essentially passes only the in-memory implementation of the service to the abstract class. All test logic is defined in AbstractTodoListServiceTest.

You can run this test by right-clicking **InMemoryTodoListServiceTest** → **Run As** → **JUnit Test**. It should finish without errors.

3.2.5 The todo-service-mongodb-impl project

The *todo-service-mongodb-impl* project demonstrates how to use the Liberty built-in mongodb-2.0 feature together with CDI to implement the todo-service. It also demonstrates how to use Arquillian and embedded mongodb to do integration test against the Liberty profile server automatically.

Introduction to the todo-service-mongodb-impl project

The todo-service-mongodb-impl project used MongoDB to store TodoList and TodoListItem. The Liberty profile server provides configuration support for MongoDB through the mongodb-2.0 feature. This feature enables the use of the MongoDB Java Driver and allows DB instances to be configured in the server configuration, injected into managed components such as EJBs and CDIs, and accessed through JNDI. Applications interact with these DB instances through the MongoDB APIs. To use the mongodb-2.0 feature provided by the Liberty profile server, mongodb-java-driver should not be packaged into the application. In this case, we set the scope of the mongodb dependency to `provided` in the `pom.xml` file (Figure 3-9).

```
<dependency>
   <groupId>org.mongodb</groupId>
   <artifactId>mongo-java-driver</artifactId>
   <version>2.11.3</version>
   <scope>provided</scope>
</dependency>
```

Figure 3-9 Scope definition for mogodb java driver

The MongoDB implementation (MongoDBTodoListService) is annotated with `@javax.inject.Singleton`, which flags it as a CDI singleton bean. CDI is used here to inject MongoDB instances through JNDI into the Liberty profile server (Figure 3-10).

```
@javax.inject.Singleton
public class MongoDBTodoListService implements TodoListService
{
   public static final String MONGO_DB_JNDI_NAME = "mongo/TODOLIST";

   @Resource(name = MONGO_DB_JNDI_NAME)
   private DB db;
...
```

Figure 3-10 MongoDB injection

An application that uses CDI must have a `beans.xml` file. The file can be completely empty (it has content only in certain limited situations), but it must be present. For a web application, the `beans.xml` file must be in the `WEB-INF` directory. For EJB modules or JAR files, the `beans.xml` file must be in the `META-INF` directory. In this case, an empty `beans.xml` file is created and put in `src/main/resources/META-INF` and will be packaged into the final JAR file.

Using Arquillian to test the MongoDB service implementation against Liberty profile server

Integration testing of the mongodb implementation is configured in a self-containing manner. This means that the integration test does not rely on the existence of any external resources, such as a running and preconfigured database or application server. Instead, during the execution of the integration tests, a Liberty profile test server is created and an embedded Mongo database is instantiated within the Liberty profile server and exposed through JNDI data sources.

Toolkit usage

To do integration testing against the Liberty profile server for the MongoDB implementation, we used the following toolkits:

- Arquillian

  With this innovative and highly extensible testing platform for the JVM, developers can easily create automated integration, and functional and acceptance tests for Java middleware. See the following website:

  `http://arquillian.org/`

- Arquillian WLP Extension

  This Arquillian container adapter (DeployableContainer implementation) can start and stop a local WebSphere Application Server Liberty profile process and run tests on it over a remote protocol (effectively in a different JVM). See the following web page:

  `https://docs.jboss.org/author/display/ARQ/WLP+V8.5+-+Managed`

- Embedded Mongo DB

  This provides a platform-neutral way for running MongoDB in unit tests, and can install and configure MongoDB automatically, so that you do not need to install and configure MongoDB before testing. See the following web page:

  `https://github.com/flapdoodle-oss/de.flapdoodle.embed.mongo`

Liberty profile server configuration

Because CDI, JNDI, and MongoDB are used in this back-end sample implementation, these features are enabled in the Liberty profile server. Meanwhile, because Arquillian is used for integration testing, it requires JMX support to control the Liberty profile server container and servlet support to invoke the test case. These two features must also be enabled. So, in the feature snippet (Figure 3-11), the entries are added.

```
<featureManager>
   <feature>localConnector-1.0</feature>
   <feature>servlet-3.0</feature>
   <feature>cdi-1.0</feature>
   <feature>mongodb-2.0</feature>
   <feature>jndi-1.0</feature>
</featureManager>
```

Figure 3-11 Feature list in server.xml

The dropins deployment feature is enabled and `"mbean"` is used as updateTrigger (Figure 3-12). This is required by the Arquillian WLP extension.

```
<applicationMonitor dropinsEnabled="true" updateTrigger="mbean" />
```

Figure 3-12 dropinsEnabled in the server.xml file

> **Note:** The dropins deployment feature is enabled by default in the Liberty profile server, so you can remove **`dropinsEnabled`**`="true"` also.

A `mongoDB` entry is configured for JNDI name `"mongo/TODOLIST"` and assumes that `mongoDB` instance is available at localhost:9991. It references the global shared library that will contain the MongoDB Java driver. See Figure 3-13 on page 24.

> **Note:** We cannot use a separate shared library because Arquillian installs the test WAR file using the dropins feature, which does not support the attachment of specific shared libraries to WARs deployed that way. Therefore, we put the MongoDB Java driver in the global shared library whose classes are automatically made available to all WAR files.

```
<mongoDB jndiName="mongo/TODOLIST" databaseName="todolistTest">
   <mongo id="mongoTestDb" libraryRef="global">
   <hostNames>localhost</hostNames>
   <ports>9991</ports>
   </mongo>
</mongoDB>
```

Figure 3-13 The mongodb configuration in the server.xml file

Arquillian WLP container configuration for integration testing

According to the requirement of Arquillian, the `arquillian.xml` file should be put into `src/test/resources` location. This file defines the Arquillian WLP container configuration (Figure 3-14) including target webcontainer port, Liberty binary location, and Liberty profile server profile name.

```
<container qualifier="wlp-managed-85" default="true">
   <configuration>
      <property name="wlpHome">target/liberty/wlp</property>
      <property name="serverName">defaultServer</property>
      <property name="httpPort">9080</property>
      <property name="appDeployTimeout">20</property>
      <property name="appUndeployTimeout">20</property>
   </configuration>
</container>
```

Figure 3-14 Arquillian wlp container configuration

Maven build lifecycle for integration testing in mongodb-impl project

To do integration testing, you can run the **`mvn verify`** command under the todo-service-mongodb-impl project. Figure 3-15 shows what happens.

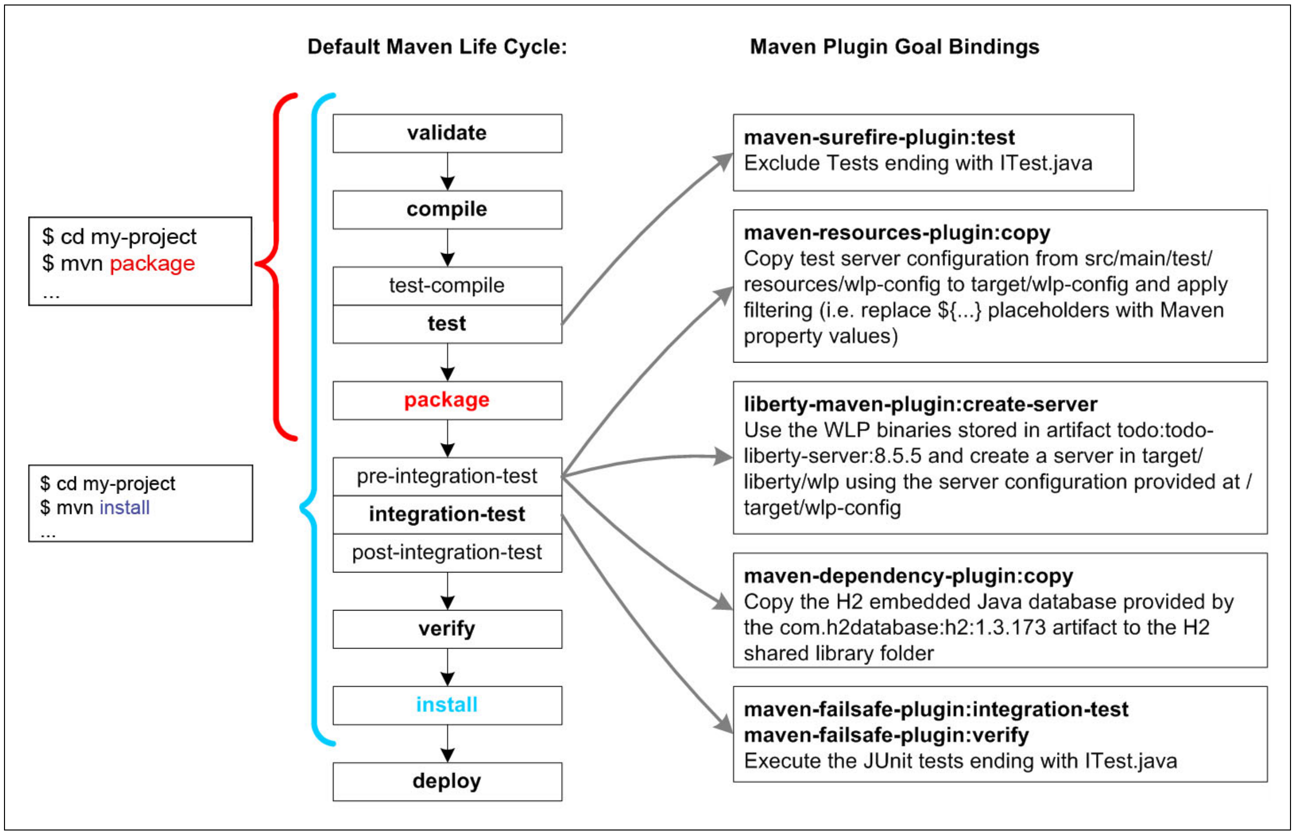

Default Maven Life Cycle:
Maven Plugin Goal Bindings
$ cd my-project
$ mvn package
...
$ cd my-project
$ mvn install
...
validate
compile
test-compile
test
package
pre-integration-test
integration-test
post-integration-test
verify
install
deploy
maven-surefire-plugin:test
Exclude Tests ending with ITest.java
maven-resources-plugin:copy
Copy test server configuration from src/main/test/resources/wlp-config to target/wlp-config and apply filtering (i.e. replace ${...} placeholders with Maven property values)
liberty-maven-plugin:create-server
Use the WLP binaries stored in artifact todo:todo-liberty-server:8.5.5 and create a server in target/liberty/wlp using the server configuration provided at /target/wlp-config
maven-dependency-plugin:copy
Copy the H2 embedded Java database provided by the com.h2database:h2:1.3.173 artifact to the H2 shared library folder
maven-failsafe-plugin:integration-test
maven-failsafe-plugin:verify
Execute the JUnit tests ending with ITest.java

Figure 3-15 Maven lifecycle for integration testing in todo-service-mongodb-impl

Various plug-ins are bound to several Maven build phases to fulfill the MongoDB implementation testing against Liberty:

- Test: maven-surefire-plugin

  By default, Maven uses maven-surefire-plugin during the test phase to run testing. It will automatically include all test classes under `src/test/java` with the following wildcard patterns:

  – Includes all of its subdirectories and all java filenames that start with `"Test"`:

    `"**/Test*.java"`

  – Includes all of its subdirectories and all java filenames that end with `"Test"`:

    `"**/*Test.java"`

  – includes all of its subdirectories and all java filenames that end with `"TestCase"`:

    `"**/*TestCase.java"`

  In the mongodb project, because the mongodb and Liberty profile server environment is not ready, running tests in this phase successfully is not possible. Therefore, testing is excluded by the snippet in `pom.xml` file (Figure 3-16 on page 26).

```
<plugin>
   <!-- exclude integration tests from surefire plugin -->
   <artifactId>maven-surefire-plugin</artifactId>
   <version>2.16</version>
   <configuration>
      <excludes>
         <exclude>**/*ITest.java</exclude>
      </excludes>
   </configuration>
</plugin>
```

Figure 3-16 Exclude testing in test phase in pom.xml

- pre-integration-test : maven-resources-plugin

  The maven-resources-plugin copies the server configuration template from the `src/test/resources/wlp-config/server.xml` file to the `target/wlp-config/server.xml` file. At the same time it filters the file, it resolves and replaces property placeholders with property values provided by the Maven run time. For the server configuration in `server.xml`, more detail is provided in "Liberty profile server configuration" on page 23.

- pre-integration-test : liberty-maven-plugin

  The liberty-maven-plugin installs the Liberty profile server binaries from the todo-liberty-server artifact to `target/liberty/wlp`. Arquillian expected this location as defined in `arquillian.xml` in the `src/test/resources` location. For details, see "Arquillian WLP container configuration for integration testing" on page 24. After executing goals in this plug-in, it creates defaultServer using the `target/wlp-config/server.xml` configuration file. The `target/wlp-config/server.xml` is copied in this location through maven-resources-plugin.

- pre-integration-test : maven-dependency-plugin

  The maven-dependency-plugin is used to copy the mongo-java-driver artifact to `wlp/usr/shared/config/lib/global/`, which is the global shared folder location:

  After the pre-integration-test phase, a Liberty profile server, defaultServer, is created in `target/liberty/wlp/usr/servers/defaultServer`. The server is preconfigured in the `server.xml` file with all needed features and settings. A Liberty profile server is ready for integration testing now.

Figure 3-17 shows the snippet for the three plugins in pre-integration-test phase.

```
<plugins>
   <plugin>
      <artifactId>maven-resources-plugin</artifactId>
      <version>2.6</version>
      <executions>
         <execution>
            <phase>pre-integration-test</phase>
            <goals>
               <goal>copy-resources</goal>
            </goals>
...
         </execution>
      </executions>
   </plugin>

   <plugin>
      <!-- before the integration test runs create a pre-configured liberty 
server -->
      <groupId>com.ibm.websphere.wlp.maven.plugins</groupId>
      <artifactId>liberty-maven-plugin</artifactId>
      <executions>
         <execution>
            <phase>pre-integration-test</phase>
            <goals>
               <goal>create-server</goal>
            </goals>
...
         </execution>
      </executions>
   </plugin>

   <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.8</version>
      <executions>
         <execution>
            <phase>pre-integration-test</phase>
            <goals>
               <goal>copy</goal>
            </goals>
...
         </execution>
      </executions>
</plugin>
```

Figure 3-17 The plug-ins in the pre-integration-test phase in the pom.xml file

- integration-test : maven-failsafe-plugin

  The maven-failsafe-plugin is used to executeJUnit test on MongoDBTodoListServiceITest test case.

  MongoDBTodoListServiceITest has several JUnit and Arquillian annotations:

- `@RunWith(Arquillian.class)`: This JUnit annotation is used to tell JUnit runs the test case through Arquillian.
- `@Deployment`: This is an Arquillian annotation and is used to give information about how to build the test WAR file. The test WAR file contains all compile and test dependencies of the mongodb-imple maven project that is achieved through the Maven.resolver() method. It also contains the MongoDBTodoListService.class, MongoDBTodoListServiceITest.class, an empty `beans.xml` to trigger CDI injection, and a `web.xml` file with a resource-env-ref definition so that JDNI lookup can work. Figure 3-18 is the snippet for `web.xml` file in the generated test WAR file.

```
<resource-env-ref>
   <resource-env-ref-name>mongo/TODOLIST</resource-env-ref-name>
   <resource-env-ref-type>com.mongodb.DB</resource-env-ref-type>
</resource-env-ref>
```

Figure 3-18 The resource-env-ref definition in web.xml

The procedure for running MongoDBTodoListServiceTest is as follows:

1. Arquillian locates and starts the Liberty profile server based on the information provided in the `src/test/resources/arquillian.xml` file
2. Arquillian deploys the test WAR file through the dropins feature and mbean trigger.
3. Arquillian triggers JUnit test execution of the MongoDBTodoListService from within the running WAR file.
4. Liberty injects the MongoDBTodoListService CDI bean implementation into the todoListService field of the test class annotated with `@Inject`. During instantiation of the MongoDBTodoListService instance, Liberty injects a mongodb JNDI resource reference in the @Resource annotated DB field. Luckily, because Liberty is all about loading and initialization, it does not complain at this time that no mongodb instance is actually running.
5. JUnit runs the setupMongoDB() method, which is annotated with the @Before JUnit annotation. This method downloads, installs, and starts an ad hoc mongodb instance on localhost:9991 in a separate process. After this step, the test environment is ready: the test WAR file is running on Liberty Server-defaultServer on port 9080, and the MongoDB instance is running on localhost:9991.
6. JUnit runs the testMongoFBTodoListService() method, which is annotated with the @Test JUnit annotation. It calls the inherited testImplementation() method and provides a reference to the CDI injected MongoDBTodoListService instance. Only now Liberty establishes a connection to the mongodb instance configured in the `server.xml`, which is running now.
7. After the test case is executed, JUnit runs the tearDownEmbeddedMongoDB() method, which is annotated with the @After JUnit annotation to stop the mongodb instance.
8. Arquillian undeploys the test WAR file, stops the Liberty profile server, and reports back to the maven-fail-safe plug-in.

3.2.6 The todo-service-jpa-impl project

The *todo-service-jpa-impl* project demonstrates how to use the Liberty profile server and JPA2 to implement the todo-service. It also demonstrates how to use Arquillian and H2 embedded database to automatically do integration testing against the Liberty profile server.

Introduction to the todo-service-jpa-impl project

The following runtime components are used for the JPA implementation:

- Hibernate-entitymanager as JPA2 implementation.
- Spring application framework for dependency injection, service discovery, and transaction management.
- Spring Data as high productivity framework for implementing JPA repositories without boilerplate. See the following web page:

  `http://www.infoq.com/articles/spring-data-intro`
- Orika as mapping framework between JPA entities and service DTOs.

The JPATodoListService is implemented as a Spring-managed service (annotated with `@org.springframework.stereotype.Service`). It uses two Spring Data repository beans that provide typical CRUD operations on the two entity types (TodoListEntity and TodoListItemEntity). The repository beans are injected by the Spring framework.

The implementation is packaged as a JAR file. Applications using this implementation must perform the typical Spring configuration for JPA-based applications.

JPA Service implementation follows a similar lifecycle, as shown in Figure 3-19, compared to other projects described in this chapter.

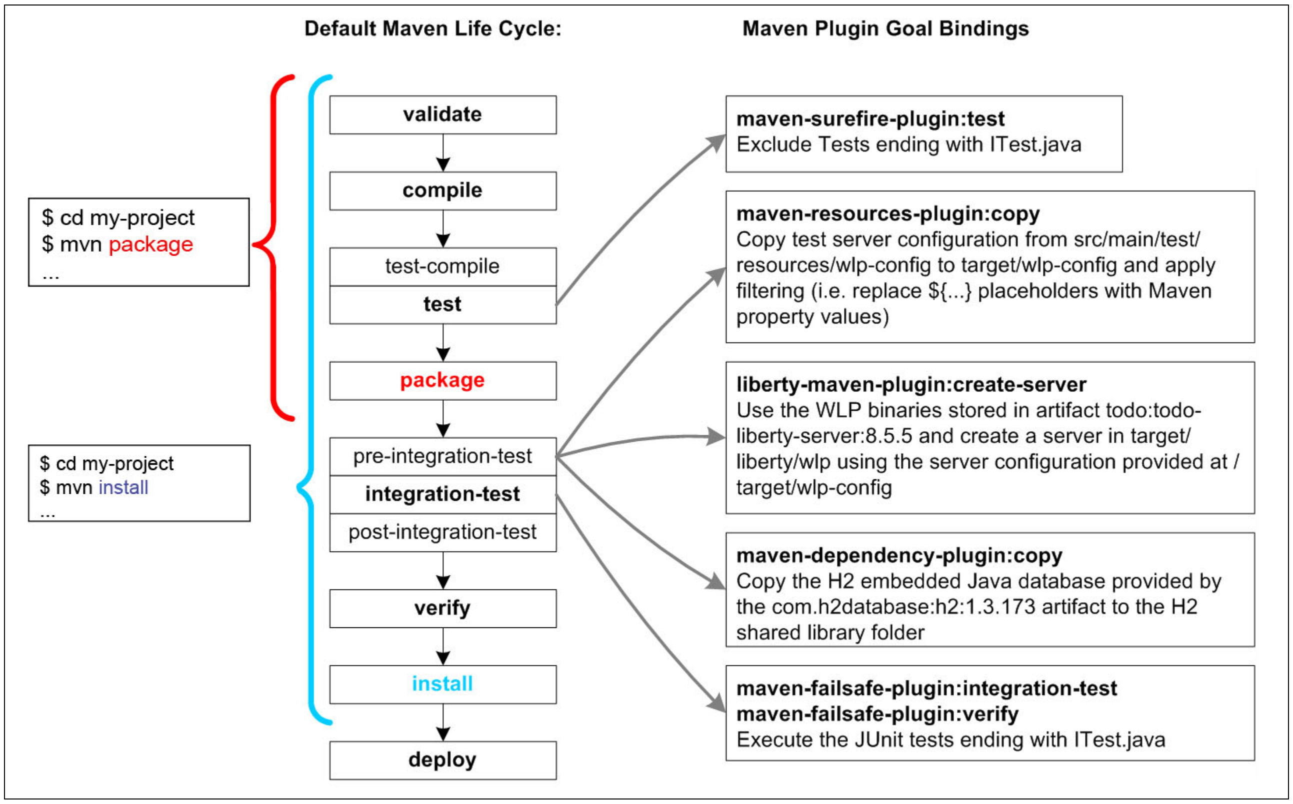

Default Maven Life Cycle:
Maven Plugin Goal Bindings
$ cd my-project
$ mvn package
...
$ cd my-project
$ mvn install
...
validate
compile
test-compile
test
package
pre-integration-test
integration-test
post-integration-test
verify
install
deploy
maven-surefire-plugin:test
Exclude Tests ending with ITest.java
maven-resources-plugin:copy
Copy test server configuration from src/main/test/resources/wlp-config to target/wlp-config and apply filtering (i.e. replace ${...} placeholders with Maven property values)
liberty-maven-plugin:create-server
Use the WLP binaries stored in artifact todo:todo-liberty-server:8.5.5 and create a server in target/liberty/wlp using the server configuration provided at /target/wlp-config
maven-dependency-plugin:copy
Copy the H2 embedded Java database provided by the com.h2database:h2:1.3.173 artifact to the H2 shared library folder
maven-failsafe-plugin:integration-test
maven-failsafe-plugin:verify
Execute the JUnit tests ending with ITest.java

Figure 3-19 Maven lifecycle for integration testing in todo-service-jpa-impl

Using Arquillian to test the JPA Service implementation against Liberty profile server

Similar to the MongoDB implementation, the integration test of the JPA implementation is configured in a self-containing manner. This means that the integration test does not rely on the existence of any external resources, such as a running and preconfigured database or application server. Instead, during the execution of the integration tests, a Liberty test server is created, and an in-memory relational database is instantiated within the Liberty profile server and exposed through JNDI data sources.

Integration testing

For our integration testing we used the following components:

- Arquillian JUnit container
- Arquillian WLP extension:

  `https://docs.jboss.org/author/display/ARQ/WLP+V8.5+-+Managed`
- H2 embedded database:

  `http://www.h2database.com/`

The `src/test/resources/wlp-config/server.xml` test-integration server configuration file is shown in Figure 3-20.

```
<?xml version="1.0" encoding="UTF-8"?>
<server description="Integration Test Server"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../todo-liberty-server/src/main/resources/server.xsd">

        <featureManager>
                <!-- JMX and Servlet support - required by Arquillian -->
<!-- https://docs.jboss.org/author/display/ARQ/WLP+V8.5+-+Managed -->
                <feature>localConnector-1.0</feature>
                <feature>servlet-3.0</feature>

                <!-- JDBC support -->
                <feature>jdbc-4.0</feature>

                <!-- JNDI Support, i.e. @Resource -->
                <feature>jndi-1.0</feature>
        </featureManager>

        <httpEndpoint id="defaultHttpEndpoint" host="localhost" httpPort="9080" />

        <library id="H2Lib">
        <fileset dir="${shared.config.dir}/lib/h2" includes="*.jar" />
        </library>

        <jdbcDriver id="H2Driver" libraryRef="H2Lib"
javax.sql.DataSource="org.h2.jdbcx.JdbcDataSource" />

        <dataSource jndiName="jdbc/TODOLIST" jdbcDriverRef="H2Driver"
type="javax.sql.DataSource">
                <connectionManager maxPoolSize="5" minPoolSize="1"
numConnectionsPerThreadLocal="1" />
        <!-- configure in-memory H2 database instance -->
                <properties URL="jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1"></properties>
        </dataSource>

        <!-- dropins support and mbean trigger required by Arquillian -->
        <applicationMonitor dropinsEnabled="true" updateTrigger="mbean" />
</server>
```

Figure 3-20 The server.xml file

Observe the following information in the `server.xml` file:

- Enabled features:
  - `localConnector-1.0`: This is JMX support, required by Arquillian to control the Liberty profile server.
  - `servlet-3.0`: This is servlet support, required by Arquillian to invoke the test case.

  – `jdbc-4.0`: This is JDBC support.
  – `jndi-1.0`: This is JNDI support so that injection of JNDI based resources (@Resource) can work.

- Webcontainer is bound to localhost:9080
- The dropins deployment feature is enabled and `mbean` is used as updateTrigger. This is required by Arquillian WLP extension
- Shared library containing the binaries of the H2 database is configured.
- JDBC driver providing access to the H2 JDBC driver is configured.
- H2 data source is configured by using the special JDBC URL prefix *`jdbc:h2:mem`*. The H2 JDBC driver is instructed not to connect to an external H2 instance but to start up an in-memory H2 instance.

The `src/test/resources/arquillian.xml` file is the Arquillian WLP container configuration. See Figure 3-21.

```
<arquillian xmlns="http://jboss.org/schema/arquillian"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://jboss.org/schema/arquillian
http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

        <engine>
                <property name="deploymentExportPath">target/</property>
        </engine>

<container qualifier="wlp-managed-85" default="true">
        <configuration>
                <property name="wlpHome">target/liberty/wlp</property>
                <property name="serverName">defaultServer</property>
                <property name="httpPort">9080</property>
                <property name="appDeployTimeout">20</property>
                <property name="appUndeployTimeout">20</property>
        </configuration>
</container>
</arquillian>
```

Figure 3-21 The src/test/resources/arquillian.xml file

For the integration test to work, a valid and complete Spring configuration is required.

Consider this information about `src/test/resources/arquillian-spring-config.xml` Spring configuration:

1. Enable annotation-based processing so that Spring creates managed instances of the implementation and the required JPA repositories. See Figure 3-22.

```
<ctx:annotation-config />
<ctx:component-scan base-package="todo.service" />
<tx:annotation-driven transaction-manager="transactionManager" />
<jpa:repositories base-package="todo.service.jpa.repository" />
```

Figure 3-22 The rc/test/resources/arquillian-spring-config.xml file

2. Look up the target (H2) data source exposed through the `server.xml`. See Figure 3-23.

```
<jee:jndi-lookup id="todolistDS" jndi-name="jdbc/TODOLIST" />
```

Figure 3-23 Target (H2) data source

3. Configure the entityManagerFactory bean that references the data source, scans the *"todo.service.jpa.model"* package for entity classes, and uses Hibernate with H2 dialect as JPA implementation. See Figure 3-24.

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
            <property name="dataSource" ref="todolistDS" />
            <property name="packagesToScan" value="todo.service.jpa.model" />
            <property name="jpaVendorAdapter">
                  <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                        <property name="generateDdl" value="true" />
                        <property name="database" value="H2" />
                        <property name="showSql" value="true" />
                  </bean>
            </property>
</bean>
```

Figure 3-24 The entityManagerFactory bean

Note: JPA is configured without the `persistence.xml` file. The key is the *"packagesToScan"* attribute. By specifying this attribute, Spring EntityManagerFactoryBean has all required metadata to bootstrap the JPA implementation.

4. Configure a Transaction Manager. See Figure 3-25.

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Figure 3-25 Transaction Manager

Based on the configuration in the `pom.xml` file, the following actions are completed on executing the integration tests with the **`mvn verify`** command:

1. The maven-resources-plugin copies the server configuration template from `src/test/resources/wlp-config/server.xml` to `target/wlp-config/server.xml`.

   At the time of copying the file, the plug-in also filters the file and resolves or replaces property placeholders with property values provided by the Maven run time.

2. The liberty-maven-plugin installs the Liberty profile server binaries from the todo-liberty-server artifact to target/liberty/wlp (the location where Arquillian expects it) and creates a *defaultServer* based using the `target/wlp-config/server.xml` configuration file.

3. The maven-dependency-plugin copies the H2 database artifact to the shared library location configured in the `server.xml` file.
4. The maven-failsafe-plugin executes JUnit on the HibernateJPATodoListServiceITest test case.

Note: Execution of tests matching `*ITest.java` is excluded for the surefire-plugin that would execute the test case in the wrong phase (test), which is before the Liberty profile server has been created.

   a. The test case is annotated with `@RunWith(Arquillian.class)` and therefore JUnit runs the test case through Arquillian.
   b. Arquillian locates the static method in the test case annotated with `@Deployment` and uses the given information to build a test WAR file. The test WAR file contains these items:
      - All compile and test dependencies of the maven project that is achieved through the Maven.resolver() function.
      - All JPA implementation classes from the `todo.service.jpa` package and its sub-packages.
      - A `web.xml` file with a resource-env-ref definition for JDNI lookup.
      - If USE_XML_BASED_SPRING_CONFIG was set to TRUE in the test case, then the XML-based Spring configuration file is included also.
      - Implicitly, the test WAR file also contains the `MongoDBTodoListServiceITest.class` file.
   c. Arquillian locates and starts the Liberty profile server based on the information provided in the `src/test/resources/arquillian.xml` file.
   d. When the Liberty profile server starts, it configures the H2 data source, which implicitly results in the startup of the in-memory H2 database.
   e. Arquillian deploys the test WAR through the dropins feature and mbean trigger.
   f. Arquillian triggers JUnit test execution of the HibernateJPATodoListServiceITest from within the running WAR file.
   g. JUnit executes the `@Before` annotated setupSpring() method, which bootstraps the Spring framework, depending on the value of USE_XML_BASED_SPRING_CONFIG. Spring is configured using either the XML-based or Java-based configuration.
   h. JUnit executes the `@Test testJPATodoListService()` method, which calls the inherited test implementation and provides a reference to the Spring-managed JPATodoListService instance.
   i. Arquillian undeploys the test WAR file, stops the Liberty profile server and reports back to the maven-fail-safe-plugin.

Integration tests can also be triggered from within Eclipse by using **Run As** → **JUnit Test**. However, the Liberty profile server must be created in the `target/liberty/wlp` directory before running the JUnit tests.

Run the **`mvn pre-integration-test`** command, which invokes all the plug-in goals that are bound to the pre-integration-test phase, eventually creating the preconfigured Liberty profile server.

4

Continuous integration with Jenkins on Liberty

Continuous integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily, leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Jenkins is one open source tool to perform continuous integration. The basic functionality of Jenkins is to monitor a version control system and to start and monitor a build system (for example Apache Ant or Maven) if changes occur. Jenkins monitors the whole build process and provides reports and notifications to alert maintainers on success or errors.

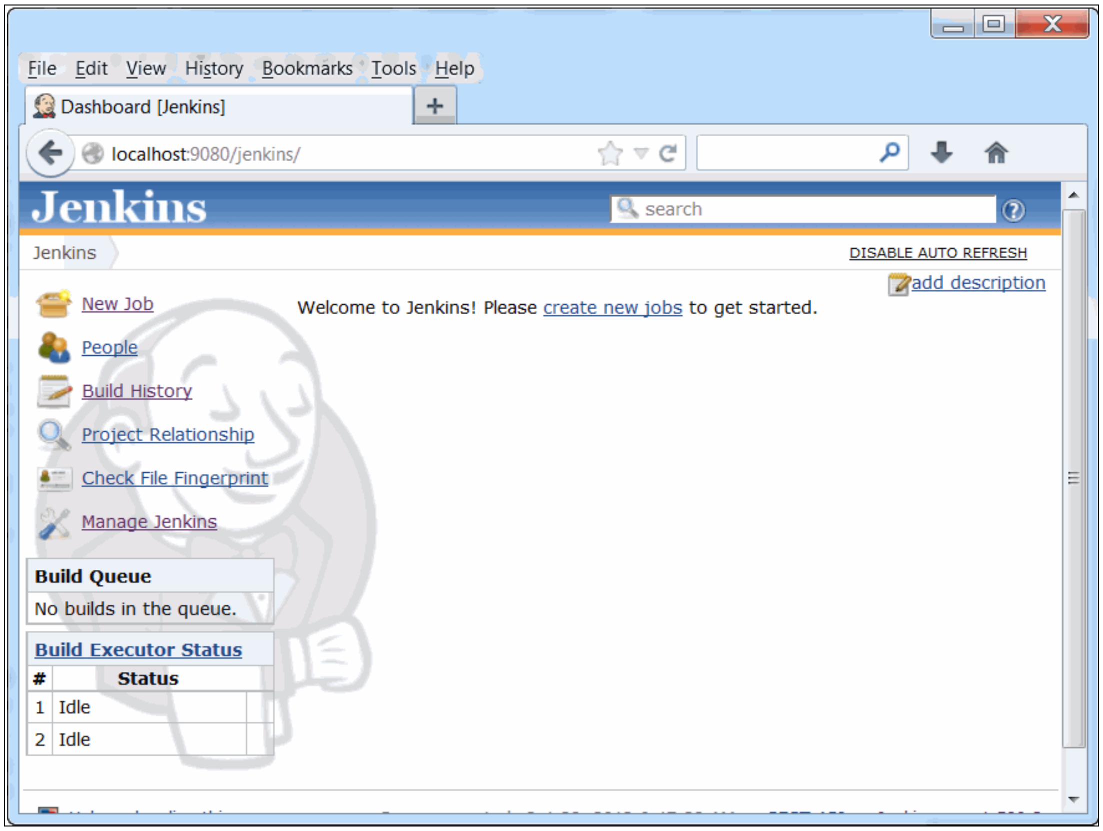
Jenkins works on all released versions of the Liberty profile server, however it works best with the latest version WebSphere Application Server v8.5.5. This chapter helps you build a continuous integration environment with Jenkins on Liberty v8.5.5.

4.1 Install Jenkins on Liberty profile server

Jenkins is available for download as native package for different OS and as Java web archive (WAR). The Java web archive can be installed on any servlet container that supports Servlet 2.4/JSP 2.0. Jenkins can be deployed on the Liberty profile server in various ways:

- Dropins

  To install Jenkins on the Liberty profile server, copy the `jenkins.war` file to the `${server.config.dir}/dropins` location; then access `http://yourhost:9080/jenkins` (see Figure 4-1 on page 36).



Welcome to Jenkins! Please create new jobs to get started.

Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Figure 4-1 Jenkins home page

- Configured

  To install Jenkins on Liberty profile server, copy the `jenkins.war` file to the `${server.config.dir}/apps` location.

  Then, edit the `server.xml` file to add the following text:

  ```
  <webApplication location="jenkins.war" />
  ```

4.1.1 Setting JENKINS_HOME

Before starting Liberty profile server, create a `${server.config.dir}/jvm.options` file and add the line shown in Figure 4-2.

```
-DJENKINS_HOME=/path/to/jenkins_home/
```

Figure 4-2 Set JENKINS_HOME

This file can also be used to increase the heap size by adding the line shown in Figure 4-3.

```
-Xmx512m
```

Figure 4-3 Set JVM Heap Size Parameter

Now start Liberty profile server and access Jenkins using the following URL format:

```
http://yourhost:9080/jenkins/
```

4.1.2 Securing Jenkins on Liberty profile server

To configure the user and group to role mappings in Liberty profile server, deploy it through the `server.xml` file. Users and groups can be picked up from LDAP, or can be stored in the `server.xml` file. Complete the following steps:

1. Configure security by enabling the appSecurity-2.0 feature. This is done by editing the `server.xml` file to add the feature. See Figure 4-4.

```
<featureManager>
   <feature>appSecurity-2.0</feature>
</featureManager>
```

Figure 4-4 Add security feature to Liberty

2. Configure the role mapping for the Jenkins application. See Figure 4-5.

```
<webApplication location="jenkins.war">
  <application-bnd>
    <security-role name="admin">
      <user name="jenkins-admin"/>
      <group name="jenkins-admins"/>
    </security-role>
  </application-bnd>
</webApplication>
```

Figure 4-5 Configure role mapping in Liberty

3. Optional: Configure the user in the `server.xml` file. See Figure 4-6.

```
<basicRegistry realm="jenkins">
  <user name="jenkins-admin" password="secret"/>
  <group name="jenkins-admins">
    <member name="jenkins-admin">
  </group>
</basicRegistry>
```

Figure 4-6 Configure users in Liberty

4. The **`securityUtility`** script in the bin folder can be used to generate a hashed password for this file. This can be done by running the command in Figure 4-7.

```
${wlp.install.dir}/bin/securityUtility encode --encoding=hash secret
```

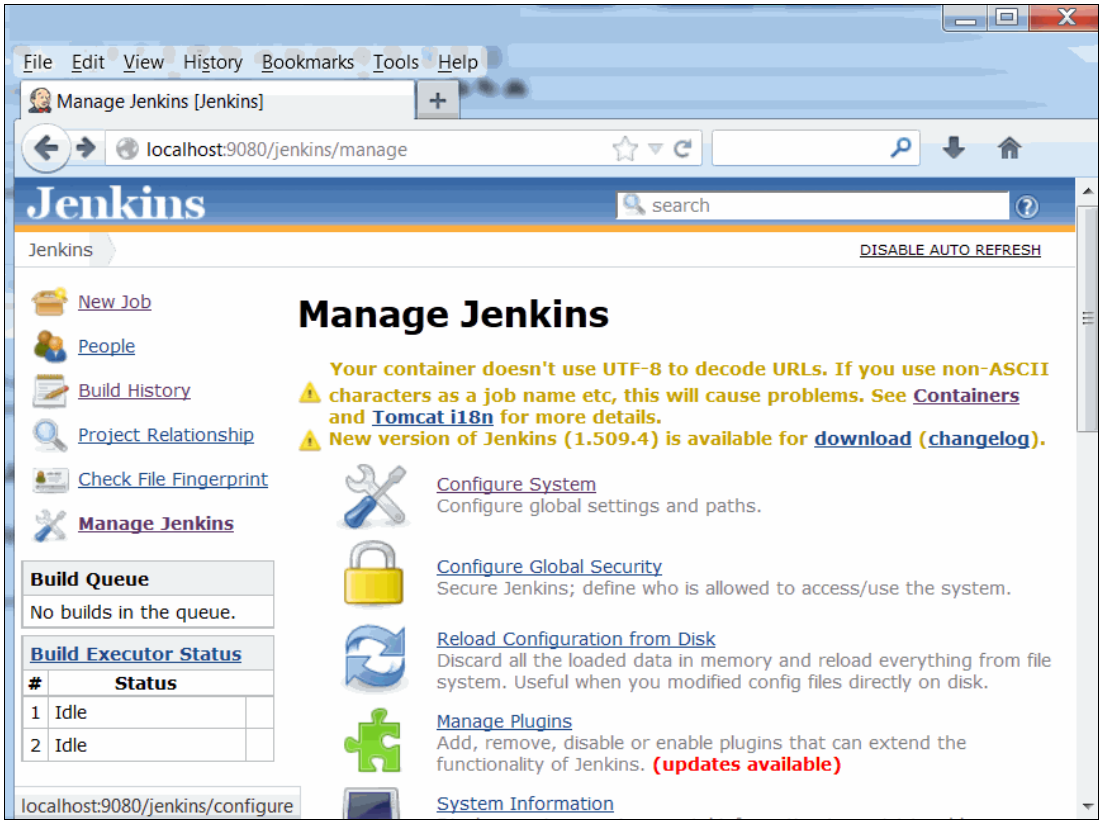
Figure 4-7 SecurityUtility command

5. Copy the printed hashed password into the password attribute in the `server.xml`. If secret is not passed in on the command line, you are prompted for it.

4.1.3 Configure Jenkins

Jenkins must be configured with JDK and Maven installation paths to run the build system. Complete the following steps:

1. Open Jenkins in the browser and click **Manage Jenkins**, and then **Configure System**. See Figure 4-8.



Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

localhost:9080/jenkins/configure

Manage Jenkins

Your container doesn't use UTF-8 to decode URLs. If you use non-ASCII characters as a job name etc, this will cause problems. See Containers and Tomcat i18n for more details.

New version of Jenkins (1.509.4) is available for download (changelog).

Configure System
Configure global settings and paths.

Configure Global Security
Secure Jenkins; define who is allowed to access/use the system.

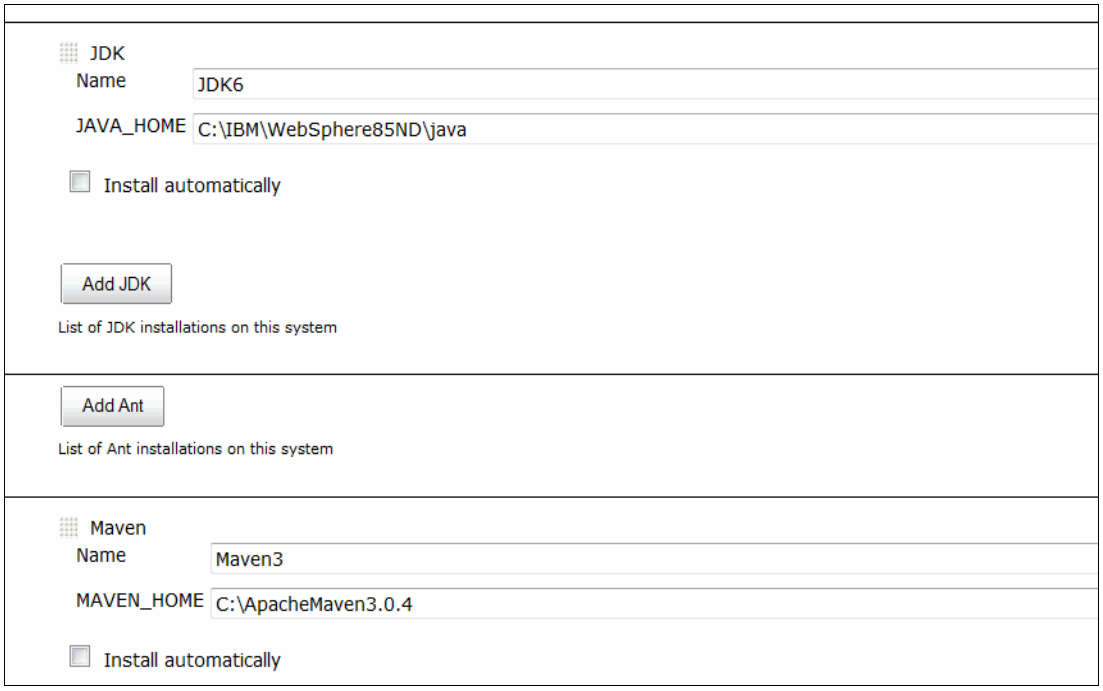
Reload Configuration from Disk
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly on disk.

Manage Plugins
Add, remove, disable or enable plugins that can extend the functionality of Jenkins. (updates available)

System Information

Figure 4-8 Manage Jenkins

2. Enter the correct path to your JDK and Maven and click **Save**. Jenkins can also install these automatically. See Figure 4-9.


JDK
Name: JDK6
JAVA_HOME: C:\IBM\WebSphere85ND\java
Install automatically

Add JDK
List of JDK installations on this system

Add Ant
List of Ant installations on this system

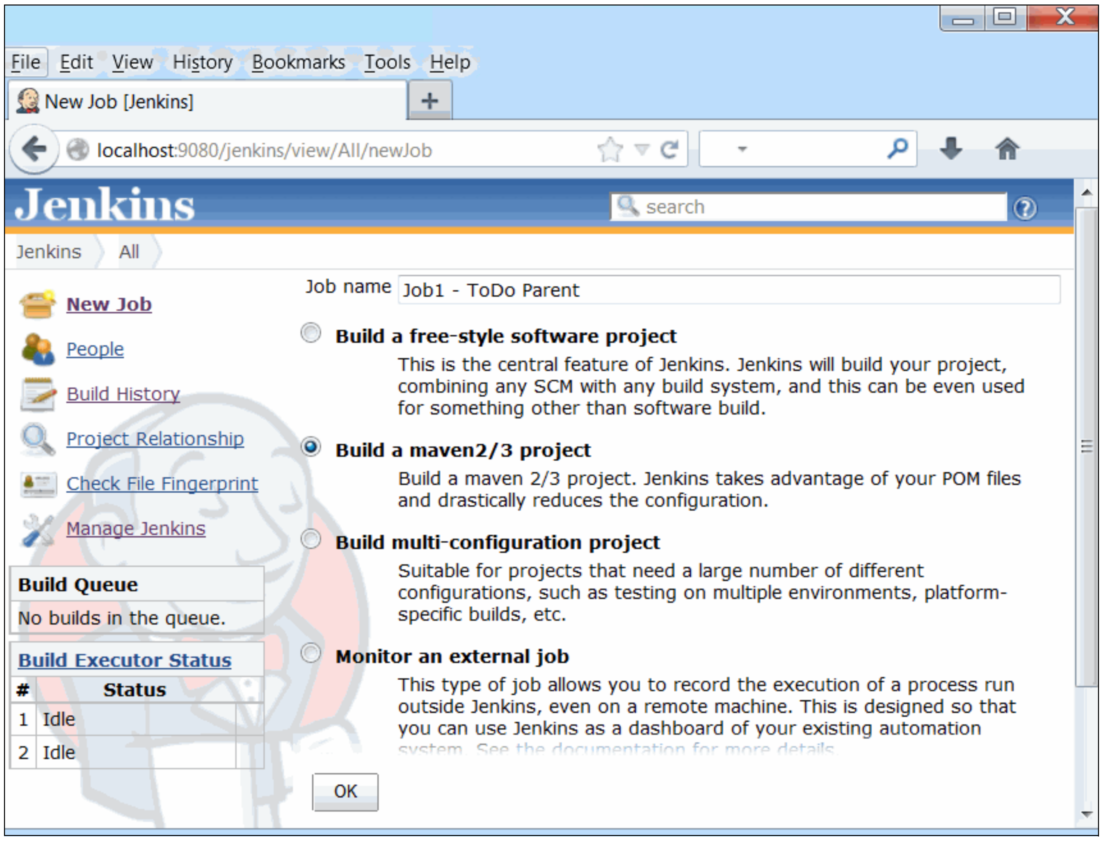
Maven
Name: Maven3
MAVEN_HOME: C:\ApacheMaven3.0.4
Install automatically

Figure 4-9 Configure JDK and Maven in Jenkins

4.2 Setting up a Jenkins job

Building a project is handled through jobs in Jenkins. Set up a job for the ToDo Project sample built using Maven by completing the following steps:

1. Click **New Job**. See Figure 4-10 on page 40.
2. Select **Build a maven2/3 project**.
3. Provide a name for the Job as **Job1 - ToDo Parent**.
4. Click **OK**.



Job name: Job1 - ToDo Parent

( ) **Build a free-style software project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

(●) **Build a maven2/3 project**
Build a maven 2/3 project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

( ) **Build multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

( ) **Monitor an external job**
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See the documentation for more details

Build Queue
No builds in the queue.

Build Executor Status

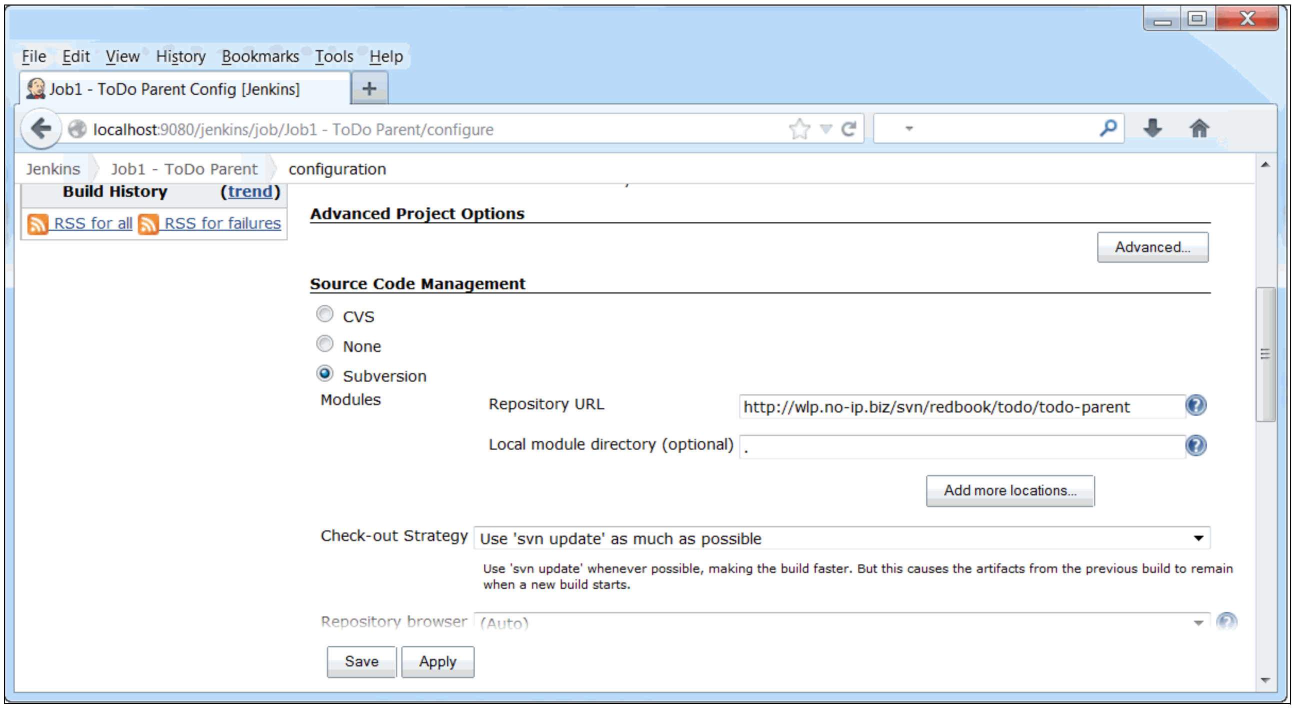
#	Status
1	Idle
2	Idle

Figure 4-10 Create Jenkins job

4.2.1 Configuring Job with Apache Subversion repository

Configure Jenkins to automatically poll the Apache Subversion repository for changes, and then create a new Jenkins build whenever changes are detected. Complete these steps:

1. Select the **Job1- ToDo Parent** job (created in Figure 4-10) and click **Configure**.
2. Select **Subversion** under the Source Code Management section. See Figure 4-11 on page 41.
3. Enter the URL of the Subversion repository you want Jenkins to poll. Click **Apply**.


File Edit View History Bookmarks Tools Help
Job1 - ToDo Parent Config [Jenkins]
localhost:9080/jenkins/job/Job1 - ToDo Parent/configure
Jenkins
Job1 - ToDo Parent
configuration
Build History
(trend)
RSS for all
RSS for failures
Advanced...
Add more locations...
Save
Apply

Advanced Project Options

Source Code Management

( ) CVS
( ) None
(●) Subversion

Modules

Repository URL: http://wlp.no-ip.biz/svn/redbook/todo/todo-parent

Local module directory (optional): .

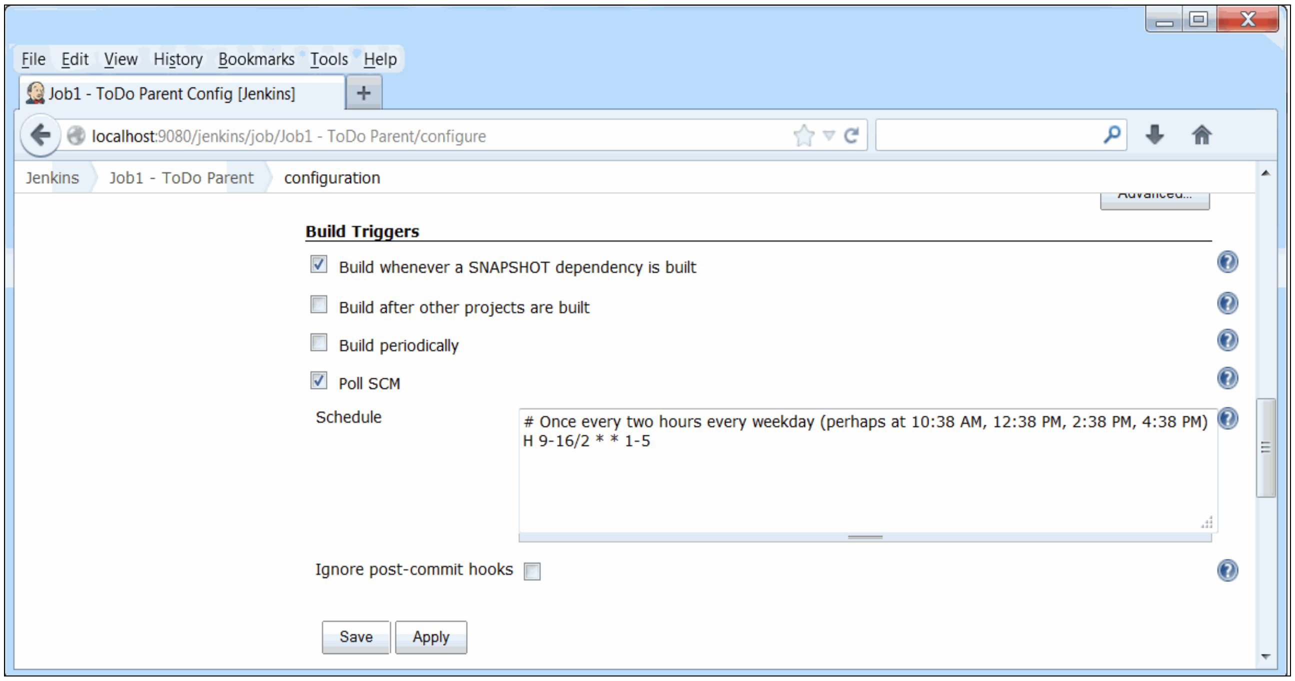
Check-out Strategy: Use 'svn update' as much as possible

Use 'svn update' whenever possible, making the build faster. But this causes the artifacts from the previous build to remain when a new build starts.

Repository browser: (Auto)

Figure 4-11 Subversion configuration

4. Under Build Triggers (Figure 4-12), select **Poll SCM.** In the Schedule text box, enter how often Jenkins should poll the Subversion repository.


File Edit View History Bookmarks Tools Help
Job1 - ToDo Parent Config [Jenkins]
localhost:9080/jenkins/job/Job1 - ToDo Parent/configure
Jenkins
Job1 - ToDo Parent
configuration
Advanced...
Save
Apply

Build Triggers

[✓] Build whenever a SNAPSHOT dependency is built
[ ] Build after other projects are built
[ ] Build periodically
[✓] Poll SCM

Schedule

```
# Once every two hours every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)
H 9-16/2 * * 1-5
```

Ignore post-commit hooks [ ]

Figure 4-12 Configure Build Triggers

5. Select **Build whenever a SNAPSHOT dependency is built** for automatic build chaining.

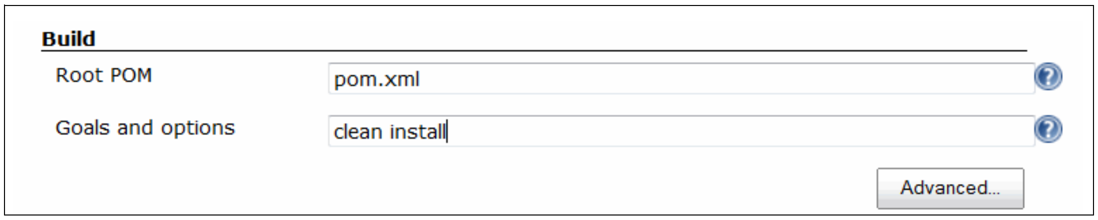
Automatic build chaining from module dependencies:

Jenkins reads dependencies of the project from the POM, and if they are also built on Jenkins, triggers are set up in such a way that a new build in one of those dependencies automatically starts a new build of the project. Jenkins understands all kinds of dependencies in POM:

- parent POM
- <dependencies> section
- <plugins> section
- <extensions> section
- <reporting> section

This process accounts for versions; you can have multiple versions or branches of your project on the same Jenkins and it correctly determines dependencies.

6. Under Build section (Figure 4-13), specify the Root POM as `pom.xml` and the Maven goals and options enter `clean install`.



Build

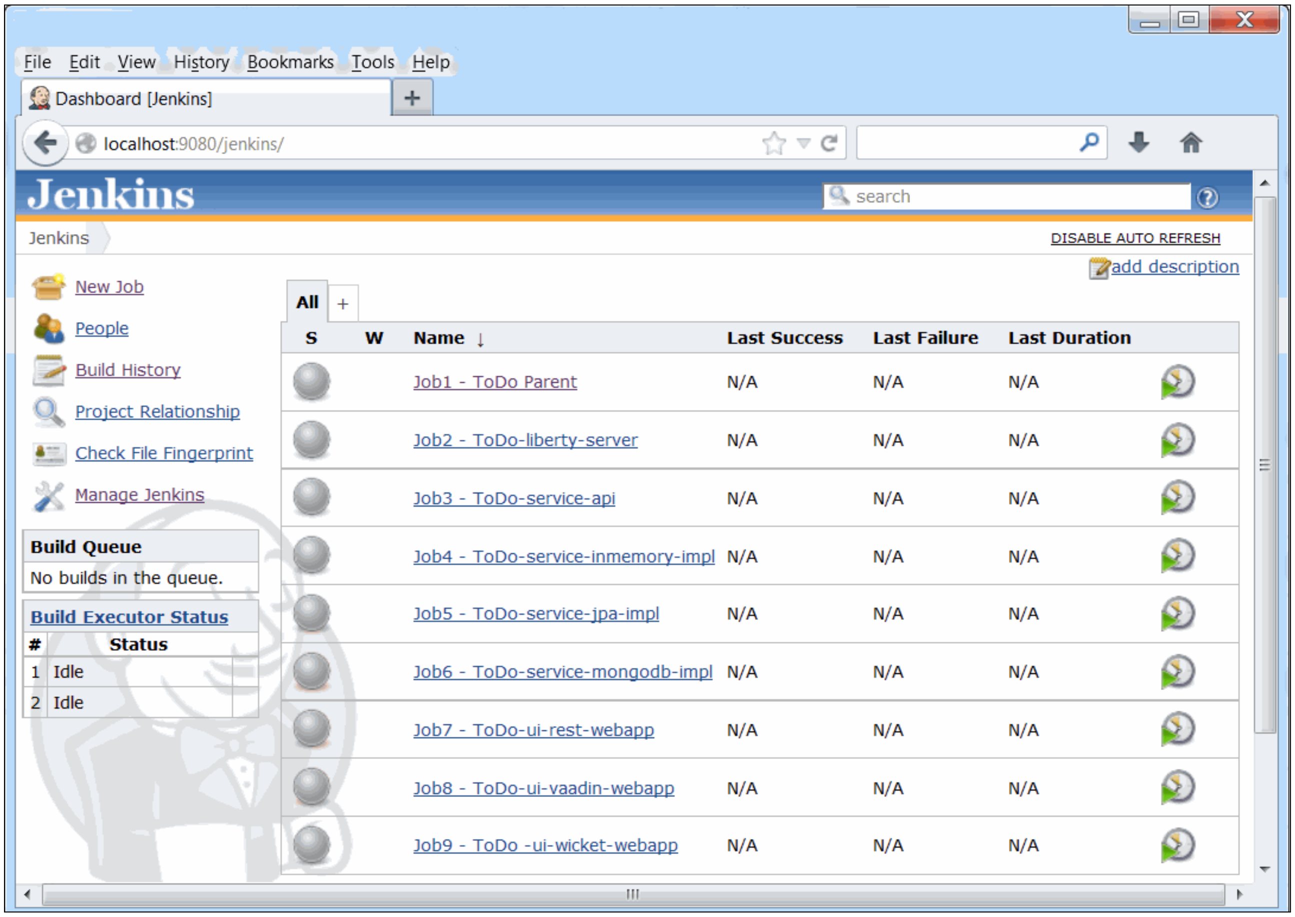
Root POM: pom.xml

Goals and options: clean install

Figure 4-13 Configure Maven Goals

7. Click **Save.**

Similarly, set up and configure jobs for the remaining modules of the ToDo Project as shown in Figure 4-14.

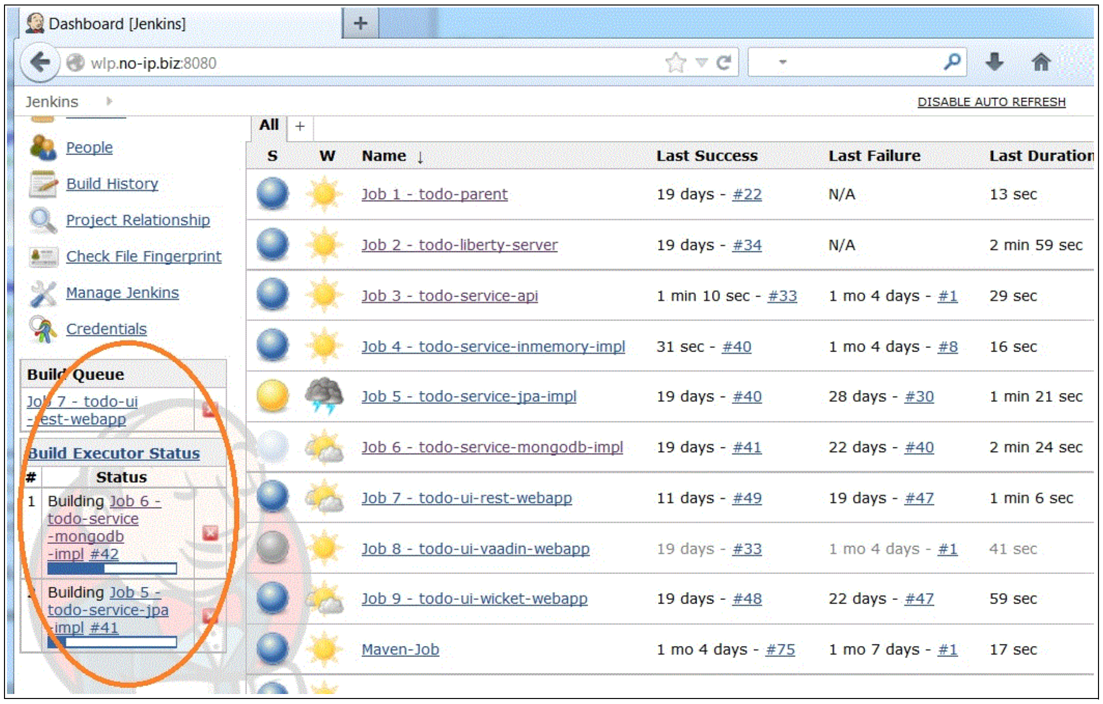


#	Status
1	Idle
2	Idle

S	W	Name ↓	Last Success	Last Failure	Last Duration
		Job1 - ToDo Parent	N/A	N/A	N/A
		Job2 - ToDo-liberty-server	N/A	N/A	N/A
		Job3 - ToDo-service-api	N/A	N/A	N/A
		Job4 - ToDo-service-inmemory-impl	N/A	N/A	N/A
		Job5 - ToDo-service-jpa-impl	N/A	N/A	N/A
		Job6 - ToDo-service-mongodb-impl	N/A	N/A	N/A
		Job7 - ToDo-ui-rest-webapp	N/A	N/A	N/A
		Job8 - ToDo-ui-vaadin-webapp	N/A	N/A	N/A
		Job9 - ToDo -ui-wicket-webapp	N/A	N/A	N/A

Figure 4-14 Setup and configure remaining modules

Jenkins instance now automatically polls the Subversion repository at the specified intervals, and will create and execute maven goals in a new build when changes are detected. Besides, Jenkins also automatically forms a build chain for any module dependencies, as shown in Figure 4-15 on page 44.


Dashboard [Jenkins]

wlp.no-ip.biz:8080

Jenkins

DISABLE AUTO REFRESH

- People
- Build History
- Project Relationship
- Check File Fingerprint
- Manage Jenkins
- Credentials

Build Queue

Job 7 - todo-ui-rest-webapp

Build Executor Status

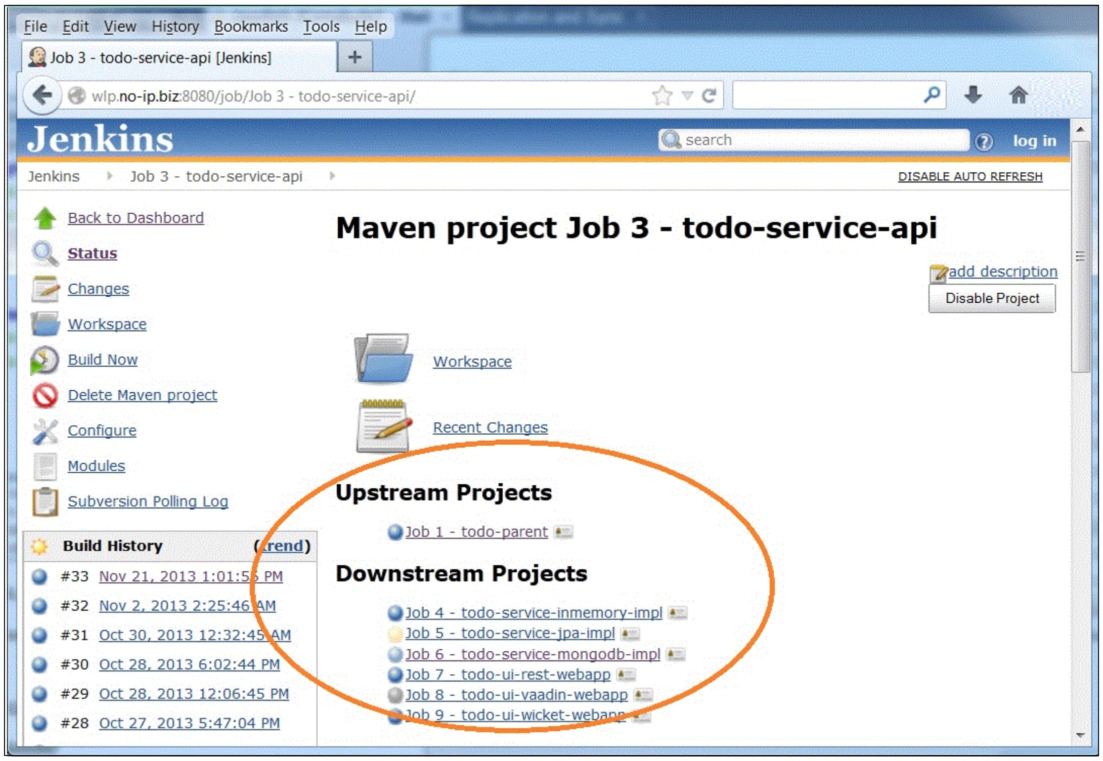
#	Status
1	Building Job 6 - todo-service-mongodb-impl #42
	Building Job 5 - todo-service-jpa-impl #41

All +

S	W	Name ↓	Last Success	Last Failure	Last Duration
●	☀	Job 1 - todo-parent	19 days - #22	N/A	13 sec
●	☀	Job 2 - todo-liberty-server	19 days - #34	N/A	2 min 59 sec
●	☀	Job 3 - todo-service-api	1 min 10 sec - #33	1 mo 4 days - #1	29 sec
●	☀	Job 4 - todo-service-inmemory-impl	31 sec - #40	1 mo 4 days - #8	16 sec
●	⛈	Job 5 - todo-service-jpa-impl	19 days - #40	28 days - #30	1 min 21 sec
○	⛅	Job 6 - todo-service-mongodb-impl	19 days - #41	22 days - #40	2 min 24 sec
●	⛅	Job 7 - todo-ui-rest-webapp	11 days - #49	19 days - #47	1 min 6 sec
●	☀	Job 8 - todo-ui-vaadin-webapp	19 days - #33	1 mo 4 days - #1	41 sec
●	⛅	Job 9 - todo-ui-wicket-webapp	19 days - #48	22 days - #47	59 sec
●	☀	Maven-Job	1 mo 4 days - #75	1 mo 7 days - #1	17 sec

Figure 4-15 Jenkins job build chain

Selecting any of the Todo jobs shows the upstream and downstream projects, as shown in Figure 4-16.


File Edit View History Bookmarks Tools Help

Job 3 - todo-service-api [Jenkins]

wlp.no-ip.biz:8080/job/Job 3 - todo-service-api/

Jenkins — search — log in

Jenkins › Job 3 - todo-service-api ›

DISABLE AUTO REFRESH

- Back to Dashboard
- Status
- Changes
- Workspace
- Build Now
- Delete Maven project
- Configure
- Modules
- Subversion Polling Log

Build History (trend)

#33	Nov 21, 2013 1:01:55 PM
#32	Nov 2, 2013 2:25:46 AM
#31	Oct 30, 2013 12:32:45 AM
#30	Oct 28, 2013 6:02:44 PM
#29	Oct 28, 2013 12:06:45 PM
#28	Oct 27, 2013 5:47:04 PM

Maven project Job 3 - todo-service-api

add description — Disable Project

Workspace

Recent Changes

Upstream Projects

- Job 1 - todo-parent

Downstream Projects

- Job 4 - todo-service-inmemory-impl
- Job 5 - todo-service-jpa-impl
- Job 6 - todo-service-mongodb-impl
- Job 7 - todo-ui-rest-webapp
- Job 8 - todo-ui-vaadin-webapp
- Job 9 - todo-ui-wicket-webapp

Figure 4-16 Upstream and downstream projects in a job

5

Front-end development on the Liberty profile server

This chapter continues to extend the sample todo list application to include two extra components: a RESTful web service with an Ajax client-side UI and a more traditional server-generated UI using Wicket.

The chapter includes the following topics:

5.1 RESTful Web Service with Ajax front end

In recent years, web application design has begun to transition away from the traditional server-side page rendering. Using Asynchronous JavaScript And XML (Ajax) technologies, modern web applications can send and receive data from servers in the background, updating interface components dynamically (without reloading the page).

5.1.1 Java Web Services on the Liberty profile server

The Liberty profile server is an ideal platform for developing and hosting modern web applications. Supporting several popular Java EE technologies, the Liberty profile server makes creating a back-end web service simple and fast.

This chapter explains implementation of the sample application component (todo-ui-rest-webapp) that uses a JAX-RS web service back end and Dojo Toolkit front end to allow browsing of todo lists and their items. This example illustrates the simplicity of using the Liberty profile server to develop a modern, dynamic, web application using the WebSphere developer tools.

Representational State Transfer (REST) refers to an architectural style of web service API design. A RESTful API design matches the HTTP request methods (GET, PUT, POST, PATCH, and DELETE) to resource URLs to expose the required functionality. Table 5-1 illustrates this concept using our sample web service component.

Table 5-1 Mapping of HTTP request methods to web service API functions

Resource (relative URLs)	**GET**	**PUT**	**POST**	**DELETE**
/items	Retrieve all items (tasks)	Replace all items with a new set	Create a new item	
/items/1	Retrieve the todo item with ID value of 1	Replace the item with a new one		Delete the item
/lists	Retrieve all todo lists	Replace all todo lists with a new set	Create a new todo list	
/lists/1	Retrieve the todo list with ID value of 1	Replace the todo list with a new one		Delete the todo list
/lists/1/items	Retrieve all items that are assigned to list with ID 1	Replace all items that belong to the list with a new set	Create a new item for this todo list	
/lists/1/items/1	Retrieve the item with ID value of 1 in the todo list with ID value of 1	Replace the todo list item with a new one		Delete the todo list item

5.1.2 Java API for RESTful web services

The Java API for RESTful Web Services, also known as JAX-RS, provides a simple mechanism for creating web service APIs with little extra code or configuration. JAX-RS extends the Java syntax to allow mapping of Java classes and methods to web service requests using the following Java annotations:

- @Path: The relative URL of the request, for example /items in Table 5-1 on page 46
- @HEAD, @GET, @PUT, @POST, @DELETE: The HTTP request type
- @Consumes: The Internet media type of the request ("application/json" for example)
- @Produces: The Internet media type of the response

In addition, to allow for request parameters to be extracted and mapped to regular Java method parameters, JAX-RS includes the following variable annotations:

- @PathParam: URL path variables (for example `/items/`**`123`**)
- @QueryParam: HTTP query parameters (for example `?searchString=`**`keywords`**)
- @Matrixparam: HTTP matrix parameters (for example `;type=`**`chore`**)
- @HeaderParam: HTTP header value (for example `Content-Length:` **`123`**)
- @CookieParam: HTTP cookie value (for example `Cookie: openList=`**`main`**)
- @FormParam: submitted form field value (for example `name=`**`go+shopping`**)
- @DefaultValue: A default value to use when the parameter is not specified
- @Context: Allows lookup of request context information such as the authenticated user, the full URI of the request or whether a secure connection is enabled

REST API implementation example

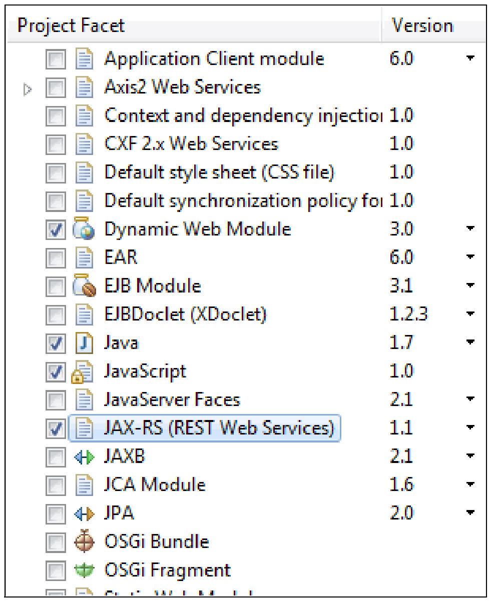
To create a RESTful web service API on the Liberty profile server, we must enable the JAX-RS feature on the Liberty profile server.

Enabling the Liberty profile server JAX-RS feature

To enable the JAX-RS feature using the WebSphere Developer Tools plug-ins for Eclipse, complete these steps:

1. Right-click the project and select **Properties**.
2. Click **Project Facets**.

3. Select the **JAX-RS (REST Web Services)** check box, as shown in Figure 5-1.



	Project Facet	Version
☐	Application Client module	6.0
☐	Axis2 Web Services	
☐	Context and dependency injectioı	1.0
☐	CXF 2.x Web Services	1.0
☐	Default style sheet (CSS file)	1.0
☐	Default synchronization policy foı	1.0
✓	Dynamic Web Module	3.0
☐	EAR	6.0
☐	EJB Module	3.1
☐	EJBDoclet (XDoclet)	1.2.3
✓	Java	1.7
✓	JavaScript	1.0
☐	JavaServer Faces	2.1
✓	JAX-RS (REST Web Services)	1.1
☐	JAXB	2.1
☐	JCA Module	1.6
☐	JPA	2.0
☐	OSGi Bundle	
☐	OSGi Fragment	

Figure 5-1 Web project facets

Enabling JAX-RS adds the required features to the Liberty profile server configuration, as shown in Figure 5-2.

```
<featureManager>
   ...
   <feature>jaxrs-1.1</feature>
   ...
</featureManager>
```

Figure 5-2 Additions to the Liberty profile server.xml configuration file

The JAX-RS application class

The application class defines the components and additional metadata configuration for a JAX-RS application. In TodoListRestApplication sample, we set the `@ApplicationPath` annotation to a value of `"api"`. This means that all requests that begin with the URL prefix of `"/api"` will be handled by this application. We also define several constants to use from our resource classes.

Note: We omit manually specifying our resource classes (in Figure 5-3) in the application because they are automatically discovered when defined in the same Java package.

```
package todo.ui.rest.jaxrs;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("api")
public class TodoListRestApplication extends Application
{
   // Static constants to use for parameters in resource request Paths
   public static final String ITEM_ID = "itemId";
   public static final String LIST_ID = "listId";
   public static final String SEARCH_STRING = "searchString";
}
```

Figure 5-3 Sample JAX-RS application

The JAX-RS resources

Our sample REST application includes two resource classes: one for to-do items (tasks) and one for to-do lists. Figure 5-4 contains a snippet from the item class, demonstrating how the JAX-RS annotations are used in practice.

The code snippet includes `@Singleton` and `@EJB` annotations to inject our concrete implementation of the back-end logic. This relationship allows for any class that implements the TodoListService interface to be used by simply modifying the build dependencies in the Maven project. This means that we can easily swap between storing the data in memory, using a MongoDB database wrapper, or even Hibernate for persisting data to a database like H2 for storage.

```
package todo.ui.rest.jaxrs.resources;
...
@Singleton
@Path("items")
@Produces(MediaType.APPLICATION_JSON)
public class Item
{
   @EJB
   private TodoListService realService;
...
   @GET
   @Path("{" + ITEM_ID + "}")
   public TodoListItem getItemById(@PathParam(ITEM_ID) final String itemId)
   {
      return realService.getItemById(itemId);
   }
...
}
```

Figure 5-4 Snippet illustrating the structure of a JAX-RS resource class

When developing your own resource classes, the WebSphere Developer Tools for Eclipse can be useful. By opting to run the project on a Liberty profile server, each time a file is changed the project files are recompiled and deployed into the running server, making changes testable instantly.

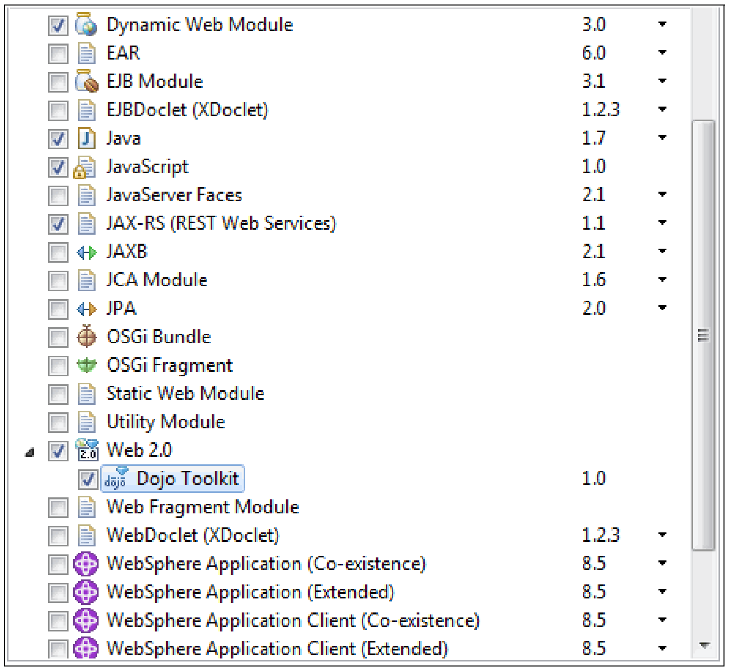
5.1.3 Dojo Toolkit

The Dojo Toolkit is an open source JavaScript framework that began development in 2004. To support and encourage adoption of Dojo, the non-profit Dojo Foundation organization was created, receiving support from several organizations including IBM.

Compared to using straight JavaScript, Dojo offers an extensive array of utility functions, UI widgets, and tools for packaging, testing, and documentation. This functionality is organized into four main components:

- Dojo: Core functionality and utility libraries that ensure consistent behavior in all supported web browsers (a time consuming issue when trying to support older platforms)
- Dijit: UI widget and theme library
- Dojox: Collection of newer and experimental modules that are not considered stable enough for inclusion in the core Dojo component
- Util: Tools for performing optimization, building (minifying and packaging), documentation and testing

Dojo is available as part of the WebSphere Developer Tools for Eclipse. When installed, Dojo can be configured from the Eclipse project *properties* menu, under the *Project Facets* section (see Figure 5-5). From there, the Dojo JavaScript files can be configured to be downloaded into your project, referenced from an existing location on disk, or from a public CDN. Our sample project uses Dojo from the Google CDN to avoid the need to package or build the source files in our illustration.



✓	Dynamic Web Module	3.0
☐	EAR	6.0
☐	EJB Module	3.1
☐	EJBDoclet (XDoclet)	1.2.3
✓	Java	1.7
✓	JavaScript	1.0
☐	JavaServer Faces	2.1
✓	JAX-RS (REST Web Services)	1.1
☐	JAXB	2.1
☐	JCA Module	1.6
☐	JPA	2.0
☐	OSGi Bundle	
☐	OSGi Fragment	
☐	Static Web Module	
☐	Utility Module	
✓	Web 2.0	
✓	Dojo Toolkit	1.0
☐	Web Fragment Module	
☐	WebDoclet (XDoclet)	1.2.3
☐	WebSphere Application (Co-existence)	8.5
☐	WebSphere Application (Extended)	8.5
☐	WebSphere Application Client (Co-existence)	8.5
☐	WebSphere Application Client (Extended)	8.5

Figure 5-5 Enabling the Dojo Toolkit option for your web project

Dojo front-end user interface sample

The structure of our sample Dojo user interface component begins in the `index.html` source file. This is where we specify the location of our Dojo and custom application script file resources, style sheets, and the core HTML anchor nodes for placing our dynamic interface components.

For the purposes of this example, we based our interface design on the Dojox mobile theme described in the following official Dojo tutorial:

http://dojotoolkit.org/documentation/tutorials/1.9/mobile/tweetview/getting_started/

Figure 5-6 shows the body section of HTML file, which has the <div> element that will contain all of our Dojo interface components and the declaration of our application script file that creates them. We reiterate this point again: by using the WebSphere Developer Tools and a running Liberty profile server when developing an application such as this, developers can view changes as they are made by simply refreshing the page. In addition, the WebSphere Developer Tools add support for creating Dojo modules and widgets using Eclipse wizards, to help you get started quickly with premade templates.

```
...
<body>
   <!-- Container to add the todo list and item widgets to programmatically -->
   <div id="view-container">
   </div>

   <!-- Application JavaScript -->
   <script type="text/javascript" src="app/main.js"></script>
</body>
</html>
```

Figure 5-6 The body of our Dojo sample index.html file

Our `main.js` module begins by listing its dependencies. The last dependency *domReady* is a special Dojo module that causes our main module to wait until the user's browser finishes loading the Document Object Module (DOM) of the web page. It also uses a simple Dojo *topic* module to handle global operation requests in the application. This allows for a simplified dependency relationship between modules and allows operation handling to be centralized. Topic subscription works by linking a unique topic keyword to a function. This can be called by using the special topic keyword followed by any arguments to pass to the handler function, as shown in Figure 5-7.

```
require([
   'app/RestUtilities',
   'app/ViewUtilities',
   'dojo/when',
   'dojo/topic',
   'dojo/domReady!'
], function (restUtilities, viewUtilities, when, topic) {
   var rootListId = 'root';
   var rootListName = 'Todo Lists';

   // Listen for requests to create list views
   topic.subscribe('createListView', viewUtilities.createListView);

   // Create the base view containing all todo lists
   when(restUtilities.requestResources('lists'), function (items) {
      topic.publish('createListView', rootListId, rootListName, items);
   });
});
```

Figure 5-7 The main.js sample application

This `main.js` module also references our two utility modules: RestUtilities and ViewUtilities. These utilities are designed to wrap commonly used functionality, such as making a request from the server with a particular set of arguments. Figure 5-8 shows the contents of the RestUtilities module. It includes a declaration of the request path prefix to the same value (api) that our RESTful web service is mapped to, and also setting the expected content type of responses to JSON. This saves configuring these constants in each separate location that a request is made from the server. In a more fully fledged example, this might be expanded to include functions for posting data back to the server.

```
define([
   'dojo/request',
   'exports'
], function (request, exports) {
   var apiUri = 'api/';

   exports.requestResources = function (requestUri) {
      return request.get(apiUri + requestUri, {
         handleAs: 'json'
      });
   };

   return exports;
});
```

Figure 5-8 The RestUtilities.js utility module

Beginning with Figure 5-9, the subsequent two figures show the Dojo todo list UI sample running on a Liberty profile server, populated with sample data.

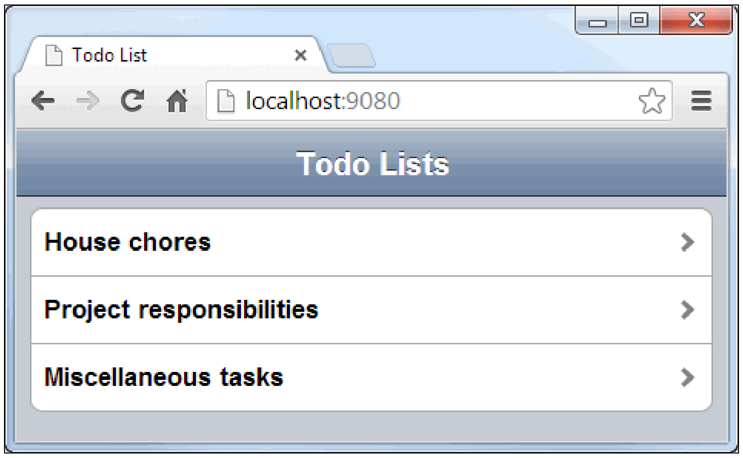


Figure 5-9 The Dojo UI home page, showing all todo lists

Figure 5-10 on page 53 and Figure 5-11 on page 53 shows a list of items for the House chores todo list. It is using the JSON returned from the JAX-RS API to set the item description, whether it is completed, and the completion deadline.

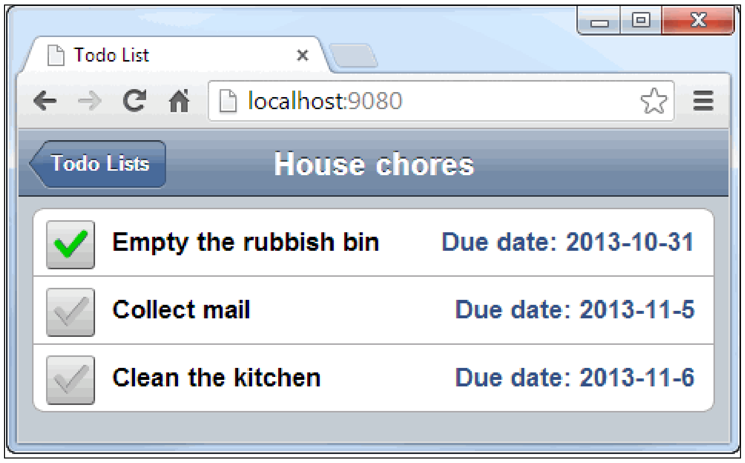


Figure 5-10 The Dojo UI todo list view, showing all the items in that list

5.1.4 Integration testing with JWebUnit on the Liberty profile server

For integration testing our web application, we use a combination of JWebUnit for UI testing and REST-assured for REST API testing. These testing tools are based on standard JUnit structure and integrate seamlessly with our continuous integration system using Jenkins.

JWebUnit

JWebUnit is a simple library for testing web application interfaces by simulating user interaction. JWebUnit provides an API for tasks such as finding text on a page, clicking on links, changing addresses, and submitting forms. Test results are based on a set of user-specified assertions, where the page elements such as page titles, text fields, links, form fields, and pop-up windows can be validated against expected values. An example of a basic test case taken from our application sample is in Figure 5-11. This test sets the web address to point to the home page, and after waiting to ensure the page has loaded, checks the title of the browser against the expected value. More information about JWebUnit is at the project home page:

http://jwebunit.sourceforge.net/

```
private WebTester wt;

@Before
public void setUp()
{
   wt = new WebTester();
   wt.setTestingEngineKey(TestingEngineRegistry.TESTING_ENGINE_HTMLUNIT);
   wt.setBaseUrl("http://localhost:9080");
}

@Test
public void testHomePage()
{
   wt.beginAt("index.html");
   waitForJavaScriptRendering(DEFAULT_WAIT_TIME);
   wt.assertTitleEquals("Todo Web Application");
}
```

Figure 5-11 Basic integration test using JWebUnit

REST-assured

REST-assured is a java-based domain-specific language (DSL) designed for testing and validating REST web services. It uses a *fluent* method chaining style, facilitating a syntax similar to languages such as Ruby. We selected this tool to enhance the coverage of the sample application integration tests. Using REST-assured, we are able to post data to the server to create lists and items then subsequently validate that the changes are reflected in the Dojo UI. The integration test from our sample application (Figure 5-12 on page 55) demonstrates how to send a POST request to the server to add a new item to a todo list, then subsequently load the page and validate that it is shown. More information about REST-assured is at the project home page:

`https://code.google.com/p/rest-assured/`

```
@Before
public void setUp()
{
   RestAssured.baseURI = "http://localhost";
   RestAssured.port = 9080;
   RestAssured.basePath = "/api";

   wt = new WebTester();
   wt.setTestingEngineKey(TestingEngineRegistry.TESTING_ENGINE_HTMLUNIT);
   wt.setBaseUrl("http://localhost:9080");
}

@Test
public void testAddList()
{
   // Check that no lists exist already
   expect().
      statusCode(200).
      body("size()", is(0)).
   when().
      get("/lists");

   // Add a test list
   given().
      contentType(MediaType.APPLICATION_JSON).
      body("{\"name\": \"test list name\", \"description\": \"test list 
description\"}").
   then().expect().
      statusCode(200).
      body(instanceOf(String.class)).
   when().
      post("/lists");

   // Check that the new list was added
   expect().
      statusCode(200).
      body("size()", is(1)).
   when().
      get("/lists");

   // Check the Dojo UI displays the new list
   wt.beginAt("index.html");
   waitForJavaScriptRendering(DEFAULT_WAIT_TIME);
   wt.assertTextPresent("test list name - test list description");
}
```

Figure 5-12 Integration test using JWebUnit and REST-assured

5.2 Apache Wicket

Another web framework we tested with the Liberty profile server is the Apache Wicket.
It is a Java-based open source web framework offering many unique features. Pages and components are just plain old Java objects (POJOs) with the markup in HTML. This means you can use any IDE and develop complex applications using encapsulation, inheritance and events. The stateless nature of the framework allows you to create complex web flows that, at any time, you can revert to the previous state of your page object. The clear separation between Java objects and corresponding markup in HTML make the development and further maintenance much easier. Page objects can be easily bound to the application data using Wicket models. Many open source libraries are available that offer easy-to-use web components like Ajax-based data grids with built-in sorting or paging, calendars, tabs, and many other components. Changing the component behavior or "look and feel" often is done by overriding the component methods so you do not need to be a JavaScript or HMTL master to use it. If you want more details about the Wicket framework, see the following website:

http://wicket.apache.org

5.2.1 Simple Todo application in Apache Wicket

Apache Wicket is a server-based Java web framework and it can easily use Spring or EJB beans. Our example uses Spring framework for Dependency Injection. Spring beans use JPA and Hibernate to permit our application data in H2 database. Our goal is to demonstrate the simple CRUD application with a bit of style by adding the Bootstrap CSS.

5.2.2 Setting up the Apache Wicket with Spring and JPA Hibernate project

You can start developing in Wicket by creating a simple Java Web Project in Eclipse and add the wicket core library or use Maven and the Wicket archetype to generate a project.

For information about setting up the Wicket project with Maven, see the following website:

http://wicket.apache.org/start/quickstart.html

Our todo application has other dependencies to todo-service-api and todo-service-jpa-impl. Both projects provide the data model we are using and the database persistence layer. To inject our database service in our Wicket web application, we use Spring Framework. To add Spring to our project we add two dependencies:

```
<dependency>
            <!-- Spring Framework -->
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
         </dependency>
         <dependency>
            <!-- Spring Support for Wicket -->
            <groupId>org.apache.wicket</groupId>
            <artifactId>wicket-spring</artifactId>
            <version>${wicket.version}</version>
         </dependency>
```

The spring-web artifact provides all necessary spring libraries we need and wicket-spring is a bride between Wicket and Spring. It allows to use the annotations in wicket pages for dependency injection. To make our pages nice-looking by enabling the Ajax check box and calendar, we add Wicket extension and date-time libraries:

```
<dependency>
         <groupId>org.apache.wicket</groupId>
         <artifactId>wicket-extensions</artifactId>
         <version>6.10.0</version>
      </dependency>
      <dependency>
         <groupId>org.apache.wicket</groupId>
         <artifactId>wicket-datetime</artifactId>
         <version>6.11.0</version>
      </dependency>
```

The complete Maven pom.xml file is available in source code of todo application. Developing in Apache Wicket does not require any specific IDE. You can use Eclipse and, alternatively in the Eclipse Marketplace, you can learn about the QWickie plug-in that helps in development.

5.2.3 Developing todo application

Having a Java web project and all necessary libraries, you can start developing the application. In Wicket, start from the application class that extends the Wicket Application abstract class. In the todo application example, it is named WicketApplication.class and it overrides two methods as shown in Example 5-1.

Example 5-1 Override

```
@Component("wicketApplication")
public class WicketApplication extends WebApplication
{
   @Override
   public Class< ? extends WebPage> getHomePage()
   {
      return ToDoListsPage.class;
   }

   @Override
   public void init()
   {
      super.init();
      getComponentInstantiationListeners().add(new SpringComponentInjector(this));
      mountBookmarkablePages();
   }
```

As the example shows, a getHomePage() method returns a ToDoListPage class; by doing this, we always get this page as a home page for our application. The second init() method is important because you might need to initialize several things when the application starts. In the sample todo application, you must initialize Spring Framework and then mount other pages.

For Spring integration, the line of code in Example 5-2 is needed.

Example 5-2 Necessary code

```
getComponentInstantiationListeners().add(new SpringComponentInjector(this));
and the @Component annotation
@Component("wicketApplication")
public class WicketApplication extends WebApplication
```

To complete the Spring framework integration with Wicket, create `applicationcontext.xml` file under the `WEB-INF` folder of the application. In this file, enable the annotation scanning for one or more code packages. In the todo list example, two packages are to be scanned.

To enable scanning, add the following two lines in the `applicationcontext.xml` file:

```
<ctx:component-scan base-package="todo.service" />
<ctx:component-scan base-package="todo.ui.wicket" />
```

The last step is to add necessary servlet and filter details in the `web.xml` descriptor file, as shown in Example 5-3

Example 5-3 Add servlet and filter details

```
<filter>
      <filter-name>wicket.WicketProject</filter-name>
      <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
      <init-param>
         <param-name>applicationFactoryClassName</param-name>
         <param-value>org.apache.wicket.spring.SpringWebApplicationFactory</param-value>
      </init-param>
      <init-param>
         <param-name>applicationClassName</param-name>
         <param-value>todo.ui.wicket.WicketApplication</param-value>
      </init-param>
   </filter>

   <filter-mapping>
      <filter-name>wicket.WicketProject</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>

   <listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
   </listener>
```

Now, you can start developing pages. The first page to be implemented is the ToDoListPage because it is a home page of the application. This class extends an AbstractBasePage class with some common code shared among all pages. This abstract class has a field of type `TodoListService` and reference name `backend` with the `@SpringBean` annotation:

```
@SpringBean
   protected TodoListService backend;
```

The `@SpringBean` annotation is from the wicket-spring library that acts as a bridge between Wicket and Spring. Based on that, Spring injects the implementation of TodoListService interface. The implementation is just a POJO class with the Spring @Service annotation located in `todo-service-jpa-impl`. Remember, Wicket is a Java framework where pages and its components are Java objects. Examine the code snippet in Example 5-4 on page 59.

Example 5-4 Code snippet

```
add(new Label("lists-count", new AbstractReadOnlyModel<Object>()
         {
            @Override
            public Object getObject()
            {
               return backend.getLists().size();
            }
         }));
```

As you can see there is a new Label object added to the page using keyword `add`. It has two parameters in its constructor.

- The first is of type String and is named `"lists-count"`. The name `"lists-count"` is later used in the HTML file corresponding to the POJO page class.
- The second is a model object that holds data to be rendered. This model is a wrapper object provided by Wicket.

See the corresponding HTML file that contains the following line:

```
   <div class="panel-heading">You currently have <span
wicket:id="items-count">2</span> items in this TODO list.</div>
```

An important step is to ensure that all your page components have unique names and HTML tags in the corresponding HTML file with the same names. In this example, the `wicket:id="items-count"` gives Wicket information where you want your label to be rendered. If a mistake in component name exists, Wicket throws an exception so every page component added to the page class must have an HTML tag with a proper `wicket:id` name.

In more complex scenarios, you must handle object hierarchy also. In the todo application, you can obtain a form for saving new todo items. The form object has two fields: name and description. Object hierarchy must be properly defined in the page class and HTML file.

The second parameter of Label object is a new instance of the Wicket's model object. Wicket provides an IModel interface and several implementations you can use while developing applications. In general, Model in Wicket is an object that wraps and holds data you want to display or edit on your page. Example 5-4 shows a new `AbstractReadOnlyModel<Object>` object as a second parameter in the label constructor. This new model object has the getModelObject() method that, in the example, returns several todo list elements. This method uses an injected Spring bean that calls the JPA hibernate and database to retrieve this value. For this simple value, you want to render the *AbstractReadOnlyModel* model implementation on your page. When you need to display and edit a value of object, you use the PropertyModel object or CompoundPropertyModel. In the `ToDoListPage.java` class you can find a more complex example of the CompoundPropertyModel used in a form object. See Example 5-5.

Example 5-5 More complex CompoundProertyModel

```
add(new Form<TodoList>("new-todo-list-form", new
CompoundPropertyModel<TodoList>(newTodoList))
         {
            {
               add(new TextField<String>("name").setRequired(true));
               add(new TextArea<String>("description"));
               final Button submitButton = new Button("submitButton")
                  {
                     @Override
                     public void onSubmit()
```

```
                              {
                                  LOG.debug("Creating new TODO list[name={}]...",
newTodoList.getName());
                                  backend.createList(newTodoList);
                                  getSession().info(newTodoList.getName() + " list has been
created.");
                                  setResponsePage(ToDoListsPage.class);
                              }
                          };
                      add(submitButton);
                  }
              });
```

The CompoundPropertyModel with the <TodoList> type now allows you to define subcomponents of type TextField andTextArea. You now use the names of the TodoList object fields as a parameters in the text field constructors. This simplifies development when handling large and complex objects that have many fields.You can use any web component offered by the Wicket framework and using names just map component to the particular field. Each type of Wicket model holds data differently. Learn more about models in the Wicket documentation:

https://cwiki.apache.org/confluence/display/WICKET/Models.

Finally, when you build and run the todo application, you are redirected to the localhost:9080 URL and the todo list page opens (Figure 5-13).

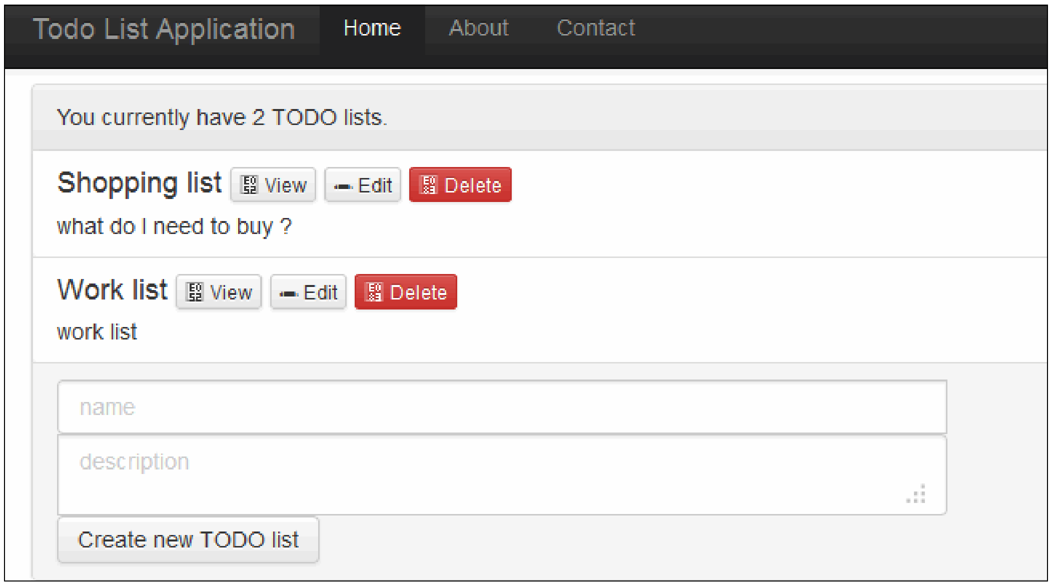
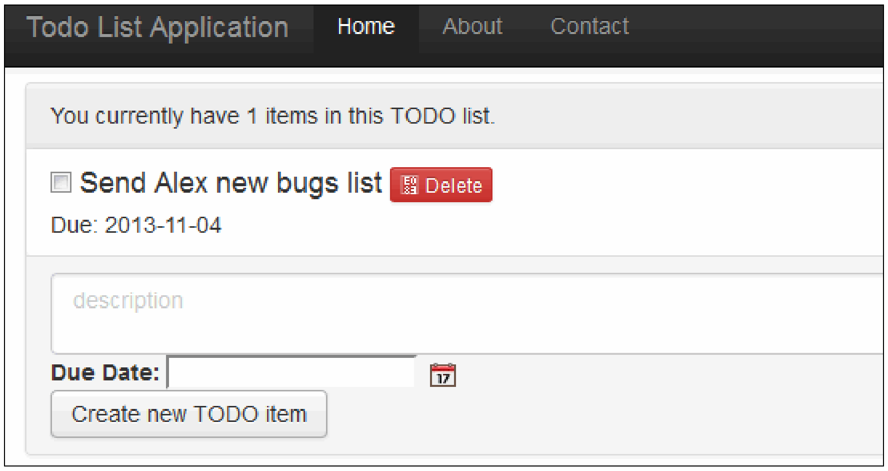


Figure 5-13 Todo List Application

You can now create todo lists into each group. See Figure 5-14. For each group with many todo items, you can manage them on the todoListItem page.



You currently have 1 items in this TODO list.

Send Alex new bugs list

Due: 2013-11-04

Figure 5-14 TODO List Application

5.2.4 Issue found during development

While we worked on the todo application code, one issue occurred on the Liberty profile server. In some cases, when you try to use the `setResponsePage(page.class)` Wicket method, you are redirected to the page but the context root of your application is missing. Assume that your todo application is deployed under the `localhost:9080/mytodo` context root path. The problem you might encounter is that the setResponsePage will redirect you to the localhost:9080 URL. A bug has already been reported on the Wicket forum but a workaround exists for this issue. To get the relative pages redirected correctly, add the code snippet from Example 5-6 to your Wicket Application class.

Example 5-6 Add this code to your Wicket Application class

```
@Override
protected WebResponse newWebResponse(final WebRequest webRequest, final
HttpServletResponse httpServletResponse)
   {
      return new ServletWebResponse((ServletWebRequest) webRequest,
httpServletResponse)
         {
            @Override
            public String encodeRedirectURL(final CharSequence relativeURL)
            {
               return new
UrlRenderer(webRequest).renderFullUrl(Url.parse(relativeURL));
            }
         };
   }
```

This code ensures Liberty profile server is working only with the absolute URLs. If you do not use the setResponsePage() method or any Ajax components in your application, you might not encounter this issue.

5.2.5 Testing Todo Wicket application

For integration and testing purposes, we use the jwebunit-htmlunit-plugin:

http://jwebunit.sourceforge.net/jwebunit-htmlunit-plugin/

The HtmlUnit is a Java-based headless browser implementation; JWebUnit is an extension providing a slick DSL language to describe test cases. Also, we use the maven-failsafe-plugin to stop the Liberty profile server after a failed build. The Wicket project `pom.xml` contains all needed dependencies and plug-in configuration. The basic concept is that we start the WebSphere Application Server Liberty profile, deploy the wicket todo application, and run the integration tests. The `server.xml` and `jvm.options` files under the `wlp-config` folder is where you can find the server configuration. The key point is that the tests that are performed using the jwebunit-htmlunit-plugin allows you to test the real HTML output of the server through HTTP. It acts as a normal browser working independently. You are able to test the HTML components, JavaScript code and simulate the user interaction with your application. Examine the test code snippet in Example 5-7

Example 5-7 Test code snippet

```
public void testWicketApplication()
{
final WebTester tester = new WebTester();
   tester.setBaseUrl("http://localhost:9080");
   tester.setTestingEngineKey(TestingEngineRegistry.TESTING_ENGINE_HTMLUNIT);
   tester.beginAt("/");
   tester.assertTitleEquals("Todo List Application");
   tester.assertTextPresent("You currently have");
   tester.assertFormPresent("new_todo_list_form1");
   // create two new todo lists
   tester.setTextField("name", "List1_Name");
   tester.setTextField("description", "List1_Description");
   tester.submit("submitButton");
   tester.setTextField("name", "List2_Name");
   tester.setTextField("description", "List2_Description");
   tester.submit("submitButton");
     tester.assertTextPresent("List1_Name");
   tester.assertTextPresent("List2_Name");
   tester.assertTextPresent("List1_Description");
   tester.assertTextPresent("List2_Description");
```

Developing complex test scenarios independently of any web framework you might have used can be easy to do. In this simple todo application example, we test only the title of todo list page, and ensure that there are some HTML components present with certain names. Finally, we add a test todo list in a form, test the save button, and ensure we successfully added new todo list.

We have one more item to mention in the `server.xml` configuration file: the Maven placeholder variable `${project.build.directory}/${project.build.finalName}`. This variable gets resolved later by the maven-resources-plugin because we do not point to the WAR file but to the exploded WAR location to improve the application startup and reload performance. Now we examine the Maven configuration.

When executing integration tests by using the **`mvn verify`** command, based on our configuration in the `pom.xml`, the maven-war-plugin creates the exploded WAR directory and a packaged WAR file in the `/target` directory.

The following list describes each phase:

- The pre-integration-test phase:

  a. The maven-resources-plugin copies the server configuration template from the `src/test/resources/wlp-config/server.xml` to the `target/wlp-config/server.xml` file. At the same time, it filters the file and resolves and replaces property placeholders with property values provided by the Maven run time.

  b. The liberty-maven-plugin installs the Liberty profile server binaries from the todo-liberty-server artifact to the `target/liberty/wlp` location and creates a defaultServer based on using the configuration file located in the `target/wlp-config/server.xml` file

  c. The liberty-maven-plugin starts the Liberty profile server instance.

  d. Liberty automatically deploys the WAR file configured in the `server.xml` file

- The integration-test phase:

  a. The maven-failsafe-plugin executes JUnit on the WicketApplicationITest test case.

  b. The testWicketApplication() method instantiates an instance of JWebUnit's WebTester client, points it to the running test server at localhost:9080, and performs the user simulation.

  Important: Execution of tests matching `*ITest.java` is excluded for the surefire plugin that will execute the test case in the wrong phase (test), which is before the Liberty profile server has been created and started.

- The post-integration-test phase:

  The liberty-maven-plugin stops the Liberty profile server

This configuration can also be used for console-based deployments during application development when you do not use Eclipse and the installed Liberty profile server. If you want to create only the server, start it, and deploy the todo application, you must call **`mvn pre-integration-tests`**. This invokes all plug-ins to set up and deploy server but will skip integration tests and will not shut down the server.

If you want to make changes to the application code under `/src/main`, to get the application redeployed, use the **`mvn package`** command. This creates the web archive; because updateTrigger polling is enabled, the server realizes the change and instantly reloads the application.

If unit test cases exist, you can skip them by using **`mvn -DskipTests`** package.

To stop WebSphere Liberty profile server use **`mvn liberty:stop-server`** command.

6

Deploying Liberty profile server with Chef

This chapter describes the basic steps of how to install the Liberty profile server on a remote system and to deploy our sample application using the Opscode Chef deployment automation solution.

The chapter contains the following topics:

6.1 About Knife, Recipe, and other terms

Chef partially uses a unique terminology in an analogy to real-life cooking, and might be confusing when you are getting started:

- Knife: This command-line tool interacts with the Chef framework. It is used to create new cookbooks and configurations and to issue **`deploy`** commands.
- Resource: This is a module that knows how to configure components of a specific type. Just as you might have a resource named *blender* in your own kitchen that knows how to configure (that is, puree) vegetables and fruits, Chef has a resource named *package* that is can install and configure binary packages, and a resource named *service* that can manage services.
- Recipe: This collection of instructions must be executed in a specific order on a target node to eventually achieve a configuration state that you want. Recipes delegate to resources, which are then responsible for actually configuring the required components. Recipes might also include other recipes, either from the same or a different cookbook.
- Cookbook: This is a container and name space for one or more recipes and the associated resources, configurations and metadata.
- Kitchen: This local configuration repository holds all cookbooks, node configurations, and so on, that are required to deploy nodes in a given environment. Knife is usually executed within a Kitchen. Kitchens can be managed by version control systems.
- Cooking: With Knife (that is, using the **`knife solo cook <options>`** command) effectively means deploying configurations to one or more target nodes.

For more information, see the following resources:

- Overview of the Chef architecture and its building blocks at the following website:

  `http://docs.opscode.com/chef_overview.html`
- Chef glossary:

  `https://wiki.opscode.com/display/chef/Glossary`

6.2 Preparing for Chef

This section describes the initial preparations for working with Chef.

The standard Chef setup advocated by Opscode involves the installation and configuration of a full Chef server and the rollout of Chef agents on all target nodes. The *chef-solo* stand-alone mode allows local deployments without the need of the Chef infrastructure. Using chef-solo in conjunction with the open source add-on *knife-solo* elegantly overcomes the limitation of “local configurations only,” allowing agent-less deployment of remote nodes.

For more information, see the following resources:

- The knife-solo plug-in is available at the following address:

  `http://matschaffer.github.io/knife-solo/`
- A good introductory video to knife-solo is at the following location:

  `http://devops.mashion.net/2011/08/04/chef-solo-basics/`

6.2.1 Preparing the test machines

For the examples in this chapter to work without alteration, you must prepare two servers (for example virtual machines) with Ubuntu Server as the operating system.

Prepare two physical or virtual machines located within the same network segment with Internet connectivity and perform the following steps on *both* machines:

1. Install Ubuntu Server 13.10 server, which you can download from the following website:

   `http://www.ubuntu.com/download/server`

2. Install OpenSSH using the following command:

   ```
   sudo apt-get install ssh
   ```

3. Create a new user named `chef`, on both machines, by using the following command:

   ```
   sudo add user chef
   ```

4. Optional: Grant password-less execution of **`sudo`** commands to the chef user by adding the following line to the `/etc/sudoers` file:

   ```
   chef ALL=NOPASSWD: ALL
   ```

   Note: Although granting this permission is optional, when not granted you might receive password prompts during deployments.

Change the host name of the *first* machine to `chefmaster` and enable SSH connectivity to the *second* machine:

1. Edit the `/etc/hostname` file and change `ubuntu` to `chefmaster`.
2. Edit the `/etc/hosts` file to change `ubuntu` to `chefmaster` and add a new line with the IP address of the second machine and the host name `chefslave`, as in this example:

   ```
   192.168.140.3 chefslave
   ```

3. Reboot the machine.
4. Verify that you can access the slave machine through SSH using chefslave as host name:

   ```
   ssh chef@chefslave
   ```

Change the host name of the *second* machine to `chefslave` and enable SSH connectivity to the *first* machine:

1. Edit the `/etc/hostname` file and change `ubuntu` to `chefslave`.
2. Edit the `/etc/hosts` file, change `ubuntu` to `chefslave` and add a new line with the IP address of the second machine and the host name `chefmaster`, as in this example:

   ```
   192.168.140.2 chefmaster
   ```

3. Reboot the machine.
4. Verify that you can access the master machine through SSH using chefmaster as host name:

   ```
   ssh chef@chefmaster
   ```

6.2.2 Installing Chef

After setting up the initial machine, you can now install Chef on the master node. Log in to chefmaster and do the following actions on the command line:

1. Configure password-less SSH Public Key authentication for logins to chefslave using the **`ssh-keygen`** and **`ssh-copy-id`** commands (Example 6-1).

Example 6-1 Configuring SSH Public Key authentication

```
chef@chefmaster:~$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/home/chef/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/chef/.ssh/id_rsa.
Your public key has been saved in /home/chef/.ssh/id_rsa.pub.
...
chef@chefmaster:~$ ssh-copy-id chef@chefslave
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any
that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now
it is to install the new keys
chef@chefslave's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'chef@chefslave'"
and check to make sure that only the key(s) you wanted were added.

chef@chefslave:~$ ssh chef@chefslave
Welcome to Ubuntu 13.10 (GNU/Linux 3.11.0-12-generic i686)

 * Documentation:  https://help.ubuntu.com/
chef@chefslave:~$
```

Note: Even after setting up password-less SSH authentication, you might still be prompted for a password when executing knife solo commands because some of them rely on the execution of Linux commands through **`sudo`**. To avoid this, add a corresponding change to the sudo configuration.

2. Install Chef using Opscode's multiplatform installer, *Omnibus installer*. See Example 6-2.

Example 6-2 Installing Chef with the Omnibus Installer

```
chef@chefmaster:~$ curl -L https://www.opscode.com/chef/install.sh | sudo bash
Downloading Chef for ubuntu...
Installing Chef
Selecting previously unselected package chef.
(Reading database ... 56401 files and directories currently installed.)
Unpacking chef (from .../tmp.r4FvBRuY/chef__i386.deb) ...
Setting up chef (11.8.0-1.ubuntu.13.04) ...
Thank you for installing Chef!
```

If the **`curl`** command is not installed, you can either install it using **`sudo apt-get install curl`** or use **`wget`** instead as follows:

```
wget -qO- https://www.opscode.com/chef/install.sh | sudo bash
```

3. Verify that the Chef binaries are installed correctly using the commands in Example 6-3.

Example 6-3 Chef installation verification

```
chef@ubuntu:~$ tree -L 1 /opt/chef/
/opt/chef/
··· bin
··· embedded
··· version-manifest.txt

2 directories, 1 file

chef@chefmaster:~$ knife -v
Chef: 11.8.0
```

4. Install knife-solo, which adds the **solo** subcommand to the **knife** command-line tool. After issuing the command be patient, it might take some time until you see any progress in the console. See Example 6-4.

Example 6-4 Installing knife-solo

```
chef@chefmaster:~$ sudo /opt/chef/embedded/bin/gem install knife-solo
Fetching: knife-solo-0.4.0.gem (100%)
Thanks for installing knife-solo!

If you run into any issues please let us know at:
  https://github.com/matschaffer/knife-solo/issues

If you are upgrading knife-solo please uninstall any old versions by
running `gem clean knife-solo` to avoid any errors.

See http://bit.ly/CHEF-3255 for more information on the knife bug
that causes this.
Successfully installed knife-solo-0.4.0
1 gem installed
Installing ri documentation for knife-solo-0.4.0...
Installing RDoc documentation for knife-solo-0.4.0...
```

5. Configure knife with several default values. See Example 6-5. You can ignore the displayed warnings.

Example 6-5 Configuring knife

```
chef@chefmaster:~$ knife configure -r . --defaults
WARNING: No knife configuration file found
*****

You must place your client key in:
  /home/chef/.chef/chef.pem
Before running commands with Knife!

*****

You must place your validation key in:
  /etc/chef-server/chef-validator.pem
Before generating instance data with Knife!

*****
Configuration file written to /home/chef/.chef/knife.rb
```

6. Create a new configuration repository called **mychefrepo** in the user's home directory using the **`knife solo init <reponame>`** (see Example 6-6):

Example 6-6 Creating a configuration repository

```
chef@chefmaster:~$ knife solo init mychefrepo
WARNING: No knife configuration file found
Creating kitchen...
Creating knife.rb in kitchen...
Creating cupboards...

chef@chefmaster:~$ tree -a mychefrepo/ | grep -v .git
mychefrepo/
··· .chef
·   ··· knife.rb
··· cookbooks
··· data_bags
··· environments
··· nodes
··· roles
··· site-cookbooks

7 directories, 8 files
```

7. Use the **`cd`** command (Example 6-7) to change to the newly created repository and execute the **`knife solo prepare chef@chefslave`** command, which installs Chef on the remote slave node.

Example 6-7 Installing Chef on the remote slave node

```
chef@chefmaster:~$ cd mychefrepo
chef@chefmaster:~/mychefrepo$ knife solo prepare chef@chefslave
Bootstrapping Chef...
--2013-11-04 16:32:17--  https://www.opscode.com/chef/install.sh
Resolving www.opscode.com (www.opscode.com)... 184.106.28.83
Connecting to www.opscode.com (www.opscode.com)|184.106.28.83|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6790 (6.6K) [application/x-sh]
Saving to: ‘install.sh’

100%[======================================>] 6,790       --.-K/s   in 0s

2013-11-04 16:32:18 (1.70 GB/s) - ‘install.sh’ saved [6790/6790]

Downloading Chef 11.8.0 for ubuntu...
Installing Chef 11.8.0
Selecting previously unselected package chef.
(Reading database ... 56387 files and directories currently installed.)
Unpacking chef (from .../chef_11.8.0_i386.deb) ...
Setting up chef (11.8.0-1.ubuntu.13.04) ...
Thank you for installing Chef!
Generating node config 'nodes/chefslave.json'...
```

8. In addition to installing Chef on the slave node, the **knife solo prepare** command also creates a node configuration file under the mychefrepo/nodes/chefslave.json location, with an empty run list configuration: {"run_list":[]}. The run list describes which recipes should be executed on this node during "cooking". Even with an empty run list, the deployment process can be invoked for verification purposes. See Example 6-8.

Example 6-8 Deployment test-run

```
chef@chefmaster:~/mychefrepo$ knife solo cook chef@chefslave
Running Chef on chefslave...
Checking Chef version...
Uploading the kitchen...
Generating solo config...
Running Chef...
Starting Chef Client, version 11.8.0
Compiling Cookbooks...
Converging 0 resources
Chef Client finished, 0 resources updated
```

When you issue the **knife solo cook** command, knife-solo generates a chef-solo configuration based on the content of the mychefrepo, uploads it to the /home/chef/chef-solo directory on the remote node (chefslave), and invokes the chef-solo command on the remote node to perform the local deployment based on the currently empty run-list configuration.

6.3 Installing Liberty profile server with Chef

Now that knife-solo is installed and remote connectivity is verified, you can configure mychefrepo for the Liberty profile server deployments to the slave node.

6.3.1 Loading the wlp cookbook

To install the Liberty profile server using Chef, the wlp cookbook that contains the necessary recipes, must be loaded into the local repository. To achieve this, you first download the cookbook using the **knife cookbook site download <cookbook>** command and then extract the archives content into the mychefrepo's cookbooks directory. See Example 6-9.

Example 6-9 Installing the wlp cookbook

```
chef@chefmaster:~/mychefrepo$ cd cookbooks

chef@chefmaster:~/mychefrepo/cookbooks$ knife cookbook site download wlp
Downloading wlp from the cookbooks site at version 0.1.0 to
/home/chef/mychefrepo/cookbooks/wlp-0.1.0.tar.gz
Cookbook saved: /home/chef/mychefrepo/cookbooks/wlp-0.1.0.tar.gz

chef@chefmaster:~/mychefrepo/cookbooks$ tar -xzf wlp-0.1.0.tar.gz

chef@chefmaster:~/mychefrepo/cookbooks$ rm wlp-0.1.0.tar.gz

chef@chefmaster:~/mychefrepo/cookbooks$ tree -L 2
.
··· wlp
    ··· attributes
    ··· Berksfile
    ··· CHANGELOG.md
    ··· DEVELOPING.md
    ··· Gemfile
    ··· libraries
    ··· LICENSE
    ··· metadata.json
    ··· metadata.rb
    ··· providers
    ··· Rakefile
    ··· README.md
    ··· recipes
    ··· resources
    ··· templates
```

6.3.2 Deploying the Liberty profile server binaries

Download the Liberty binary JAR files from the following location and place them in a directory on a web server that is reachable from the test servers:

https://www.ibmdw.net/wasdev/downloads/websphere-application-server-liberty-profile/

For example, place them here:

`http://repo.local/wlp/8.5.5.0/`

The cookbook's website lists all the contained recipes and configurable attributes:

http://community.opscode.com/cookbooks/wlp

Based on this information, modify the chefslave's node configuration located at `mychefrepo/nodes/chefslave.json` as in Example 6-10:

Example 6-10 Chef slave node configuration to install Liberty profile server

```
{
   "wlp": {
      "user": "wlp",
      "group": "wlp",
      "base_dir": "/opt/ibm",
      "install_method": "archive",
      "install_java": true,
      "archive": {
         "base_url": "http://repo.local/wlp/8.5.5.0/",
         "accept_license": true,
         "extended": { "install": true },
         "extras": { "install": false }
      }
   },

   "run_list":["recipe[wlp::default]"]
}
```

This instructs chef to run the default receipt found in the wlp cookbook and parameterize its execution with the configuration parameters. The current configuration results in an installation of the Liberty profile server at `/opt/ibm/wlp` under ownership of a to-be-created user wlp. The Liberty profile server binaries are automatically downloaded from the specified IBM website during installation, including the extended content.

No cost versus license: Be aware that these binaries are provided as a no-charge option for development purposes only. If you plan to deploy Liberty profile server to production, you must purchase an appropriate license.

To install Liberty profile server, you must "cook" the slave node again using **knife**. See Example 6-11.

Example 6-11 Deploying Liberty profile server binaries - failing attempt

```
chef@chefmaster:~/mychefrepo$ knife solo cook chef@chefslave
Running Chef on chefslave...
Checking Chef version...
Uploading the kitchen...
Generating solo config...
Running Chef...
Starting Chef Client, version 11.8.0
Compiling Cookbooks...
[2013-11-04T23:40:04+01:00] ERROR: Running exception handlers
[2013-11-04T23:40:04+01:00] ERROR: Exception handlers complete
[2013-11-04T23:40:04+01:00] FATAL: Stacktrace dumped to
/var/chef/cache/chef-stacktrace.out
Chef Client failed. 0 resources updated
[2013-11-04T23:40:04+01:00] ERROR: Cookbook java not found. If you're loading java
from another cookbook, make sure you configure the dependency in your metadata
[2013-11-04T23:40:04+01:00] FATAL: Chef::Exceptions::ChildConvergeError: Chef run
process exited unsuccessfully (exit code 1)
ERROR: RuntimeError: chef-solo failed. See output above.
```

Unfortunately, the first attempt to deploy Liberty profile server failed. The reason is that the wlp cookbook has a reference to another cookbook called *java*, which must be made available to the local repository too. Several other cookbooks are also missing, which you will realize on subsequent attempts. To have the deployment succeed, you must install these cookbooks: java, aws, windows, and chef_handler. See Example 6-12.

Example 6-12 Loading required cookbooks into the repository

```
chef@chefmaster:~/mychefrepo$ cd cookbooks/

chef@chefmaster:~/mychefrepo/cookbooks$ knife cookbook site download java
Downloading java from the cookbooks site at version 1.15.4 to
/home/chef/mychefrepo/cookbooks/java-1.15.4.tar.gz
Cookbook saved: /home/chef/mychefrepo/cookbooks/java-1.15.4.tar.gz

chef@chefmaster:~/mychefrepo/cookbooks$ knife cookbook site download aws
Downloading aws from the cookbooks site at version 1.0.0 to
/home/chef/mychefrepo/cookbooks/aws-1.0.0.tar.gz
Cookbook saved: /home/chef/mychefrepo/cookbooks/aws-1.0.0.tar.gz

chef@chefmaster:~/mychefrepo/cookbooks$ knife cookbook site download windows
Downloading windows from the cookbooks site at version 1.11.0 to
/home/chef/mychefrepo/cookbooks/windows-1.11.0.tar.gz
Cookbook saved: /home/chef/mychefrepo/cookbooks/windows-1.11.0.tar.gz

chef@chefmaster:~/mychefrepo/cookbooks$ knife cookbook site download chef_handler
```

```
Downloading chef_handler from the cookbooks site at version 1.1.4 to
/home/chef/mychefrepo/cookbooks/chef_handler-1.1.4.tar.gz
Cookbook saved: /home/chef/mychefrepo/cookbooks/chef_handler-1.1.4.tar.gz

chef@chefmaster:~/mychefrepo/cookbooks$ find *.tar.gz -exec tar -xzf {} \;

chef@chefmaster:~/mychefrepo/cookbooks$ rm *.tar.gz

chef@chefmaster:~/mychefrepo/cookbooks$ ls
aws  chef_handler  java  windows  wlp
```

After the required cookbooks are installed, you can retry the deployment. See Example 6-13.

Example 6-13 Deploying Liberty profile server binaries - successful attempt

```
chef@chefmaster:~/mychefrepo$ knife solo cook chef@chefslave
Running Chef on chefslave...
Checking Chef version...
Uploading the kitchen...
Generating solo config...
Running Chef...
Starting Chef Client, version 11.8.0
Compiling Cookbooks...
Converging 19 resources
Recipe: wlp::default
  * group[wlp] action create
    - create group[wlp]

  * user[wlp] action create
    - create user user[wlp]
  * directory[/opt/ibm] action create
    - create new directory /opt/ibm
    - change mode from '' to '0755'
    - change owner from '' to 'wlp'
    - change group from '' to 'wlp'
Recipe: java::openjdk
  * package[openjdk-6-jdk] action install
    - install version 6b27-1.12.6-1ubuntu2 of package openjdk-6-jdk
  * package[openjdk-6-jre-headless] action install (up to date)
  * bash[update-java-alternatives] action run
    - execute "bash"  "/tmp/chef-script20131104-3695-15x5bnb"
Recipe: java::set_java_home
  * ruby_block[set-env-java-home] action run
    - execute the ruby block set-env-java-home
  * directory[/etc/profile.d] action create (up to date)
  * file[/etc/profile.d/jdk.sh] action create
    - create new file /etc/profile.d/jdk.sh
    - update content in file /etc/profile.d/jdk.sh from none to f7f16e
        --- /etc/profile.d/jdk.sh       2013-11-04 23:55:23.030642197 +0100
        +++ /tmp/.jdk.sh20131104-3695-1ch9v0f 2013-11-04 23:55:23.030642197 +0100
        @@ -1 +1,2 @@
        +export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-i386
    - change mode from '' to '0755'
Recipe: java::default
  * package[sun-java6-jdk] action purge (skipped due to only_if)
  * package[sun-java6-bin] action purge (skipped due to only_if)
  * package[sun-java6-jre] action purge (skipped due to only_if)
Recipe: wlp::archive_install
  * remote_file[/var/chef/cache/wlp-developers-runtime-8.5.5.0.jar] action create
    - create new file /var/chef/cache/wlp-developers-runtime-8.5.5.0.jar
    - update content in file /var/chef/cache/wlp-developers-runtime-8.5.5.0.jar
from none to 403274 (file sizes exceed 10000000 bytes, diff output suppressed)
    - change owner from '' to 'wlp'
    - change group from '' to 'wlp'
  * remote_file[/var/chef/cache/wlp-developers-extended-8.5.5.0.jar] action create
    - create new file /var/chef/cache/wlp-developers-extended-8.5.5.0.jar
    - update content in file /var/chef/cache/wlp-developers-extended-8.5.5.0.jar
from none to b3df90 (file sizes exceed 10000000 bytes, diff output suppressed)
    - change owner from '' to 'wlp'
    - change group from '' to 'wlp'
  * execute[install wlp-developers-runtime-8.5.5.0.jar] action run
    - execute java -jar /var/chef/cache/wlp-developers-runtime-8.5.5.0.jar
--acceptLicense /opt/ibm
  * execute[install wlp-developers-extended-8.5.5.0.jar] action run
    - execute java -jar /var/chef/cache/wlp-developers-extended-8.5.5.0.jar
--acceptLicense /opt/ibm

Chef Client finished, 12 resources updated
```

You can now log in to the slave node and verify the correct installation of the Liberty profile server binaries. See Example 6-14.

Example 6-14 Verifying Liberty profile server binary installation on remote node

```
chef@chefslave:~$ ls -l /opt/ibm/wlp/
total 60
drwxr-xr-x 5 wlp wlp  4096 Nov  4 17:04 bin
drwxr-xr-x 3 wlp wlp  4096 Nov  4 17:04 clients
-rw-r--r-- 1 wlp wlp   379 Nov  4 17:04 Copyright.txt
drwxr-xr-x 5 wlp wlp  4096 Nov  4 17:04 dev
drwxr-xr-x 3 wlp wlp  4096 Nov  4 17:04 lafiles
drwxr-xr-x 8 wlp wlp 20480 Nov  4 17:04 lib
-rw-r--r-- 1 wlp wlp 11331 Nov  4 17:04 README.TXT
drwxr-xr-x 3 wlp wlp  4096 Nov  4 17:04 templates
drwxr-xr-x 4 wlp wlp  4096 Nov  4 17:04 usr

chef@chefslave:~$ /opt/ibm/wlp/bin/productInfo validate
Start product validation...
Validating feature: appSecurity-1.0... PASS!
Validating feature: appSecurity-2.0... PASS!
Validating feature: beanValidation-1.0... PASS!
Validating feature: blueprint-1.0... PASS!
Validating feature: cdi-1.0... PASS!
...
Validating feature: wasJmsServer-1.0... PASS!
Validating feature: webCache-1.0... PASS!
Validating feature: webProfile-6.0... PASS!
Validating feature: wmqJmsClient-1.1... PASS!
Validating feature: wsSecurity-1.1... PASS!
Product validation completed successfully.
```

6.3.3 Creating Liberty profile server configurations

Now that the Liberty profile server binary installation is successful, you can extend the node configuration and add instructions to create two dedicated Liberty profile server configurations for the REST-based and the Wicket-based Todo List Application.

This can be achieved by using the serverconfig recipe of the wlp cookbook. This recipe expects its configuration under the `[“wlp”][“servers”]` key in the attributes map. Add the configuration outlined in Example 6-15 to the `mychefrepo/nodes/chefslave.json` file.

Example 6-15 Extended Chef slave node configuration with server creation recipe

```
{
   "wlp": {
      "user": "wlp",
      "group": "wlp",
      "base_dir": "/opt/ibm",
      "install_method": "archive",
      "install_java": true,
      "archive": {
         "base_url": "http://repo.local/wlp/8.5.5.0/",
         "accept_license": true,
         "extended": { "install": true },
         "extras": { ”install": false }
      },
      "servers": {
         "defaultServer" : { "enabled": false },
         "todo-ui-rest" : {
            "enabled": true,
            "description" : "Server for REST-based TODO list application",
            "featureManager" : {
               "feature": [ "servlet-3.0", "jaxrs-1.1", "ejbLite-3.1" ]
            },
            "httpEndpoint" : {
               "id" : "defaultHttpEndpoint",
               "host" : "*",
               "httpPort" : "9081"
            }
         },
         "todo-ui-wicket" : {
            "enabled": true,
            "description" : "Server for Wicket-based TODO list application",
            "featureManager" : {
               "feature": [ "servlet-3.0", "jdbc-4.0", "jndi-1.0" ]
            },
            "httpEndpoint" : {
               "id" : "defaultHttpEndpoint",
               "host" : "*",
               "httpPort" : "9082"
            }
         }
      }
   },

   "run_list":["recipe[wlp::default]", "recipe[wlp::serverconfig]"]
}
```

By default, the serverconfig recipe creates a new server configuration called defaultServer. To avoid that, we set the enabled option to `FALSE` for the defaultServer.

Using the feature option for each server, only those runtime features are enabled that are required by the respective application. The `wlp::serverconfig` recipe is added to the end of the run_list attribute value. In this way, the recipe is executed after the `wlp::default` recipe, which is responsible for installing the Liberty profile server binaries.

After you update the node configuration, run the **`knife solo cook`** command again (Example 6-16).

Example 6-16 Output of Liberty profile server creation with knife-solo

```
chef@chefmaster:~/mychefrepo$ knife solo cook chef@chefslave
Running Chef on chefslave...
Checking Chef version...
Uploading the kitchen...
Generating solo config...
Running Chef...
Starting Chef Client, version 11.8.0
Compiling Cookbooks...
Converging 20 resources
Recipe: wlp::default
  * group[wlp] action create (up to date)
  * user[wlp] action create (up to date)
  * directory[/opt/ibm] action create (up to date)
Recipe: java::openjdk
  * package[openjdk-6-jdk] action install (up to date)
  * package[openjdk-6-jre-headless] action install (up to date)
  * bash[update-java-alternatives] action run
    - execute "bash"  "/tmp/chef-script20131105-8613-u7xnx"
Recipe: java::set_java_home
  * ruby_block[set-env-java-home] action run
    - execute the ruby block set-env-java-home
* directory[/etc/profile.d] action create (up to date)
  * file[/etc/profile.d/jdk.sh] action create (up to date)
Recipe: java::default
  * package[sun-java6-jdk] action purge (skipped due to only_if)
  * package[sun-java6-bin] action purge (skipped due to only_if)
  * package[sun-java6-jre] action purge (skipped due to only_if)
Recipe: wlp::archive_install
  * remote_file[/var/chef/cache/wlp-developers-runtime-8.5.5.0.jar] action create
(skipped due to not_if)
  * remote_file[/var/chef/cache/wlp-developers-extended-8.5.5.0.jar] action create
(skipped due to not_if)
  * execute[install wlp-developers-runtime-8.5.5.0.jar] action run (skipped due to
not_if)
  * execute[install wlp-developers-extended-8.5.5.0.jar] action run (skipped due
to not_if)
Recipe: wlp::serverconfig
  * directory[/opt/ibm/wlp/usr/servers/todo-ui-rest] action create
    - create new directory /opt/ibm/wlp/usr/servers/todo-ui-rest
    - change mode from '' to '0775'
    - change owner from '' to 'wlp'
    - change group from '' to 'wlp'
  * wlp_config[/opt/ibm/wlp/usr/servers/todo-ui-rest/server.xml] action create
  * directory[/opt/ibm/wlp/usr/servers/todo-ui-wicket] action create
    - create new directory /opt/ibm/wlp/usr/servers/todo-ui-wicket
    - change mode from '' to '0775'
    - change owner from '' to 'wlp'
    - change group from '' to 'wlp'
  * wlp_config[/opt/ibm/wlp/usr/servers/todo-ui-wicket/server.xml] action create

Chef Client finished, 6 resources updated
```

Deploying a fresh node with this configuration can result in the installation of Java, the Liberty binaries, and eventually in the creation of the server configurations. Because all operations in Chef are independent, deploying a slave node (which already has Java and the Liberty profile server binaries installed from the previous deployment) creates only the two server configurations. Execution of the other recipes result in no system changes.

You can now verify the created server configurations on the slave node (Example 6-17).

Example 6-17 Verification of the created server configurations on the slave node

```
chef@chefslave:~$ tree /opt/ibm/wlp/usr/servers/
/opt/ibm/wlp/usr/servers/
··· todo-ui-rest
·   ··· server.xml
··· todo-ui-wicket
    ··· server.xml

2 directories, 2 files
chef@chefslave:~$ cat /opt/ibm/wlp/usr/servers/todo-ui-rest/server.xml
<server description="Server for REST-based TODO list application">
  <featureManager>
    <feature>servlet-3.0</feature>
    <feature>jaxrs-1.1</feature>
    <feature>ejbLite-3.1</feature>
  </featureManager>
  <httpEndpoint id="defaultHttpEndpoint" host="*" httpPort="9081"/>
</server>

chef@chefslave:~$ cat /opt/ibm/wlp/usr/servers/todo-ui-wicket/server.xml
<server description="Server for Wicket-based TODO list application">
  <featureManager>
    <feature>servlet-3.0</feature>
    <feature>jdbc-4.0</feature>
    <feature>jndi-1.0</feature>
  </featureManager>
  <httpEndpoint id="defaultHttpEndpoint" host="*" httpPort="9082"/>
</server>
```

6.3.4 Starting Liberty profile servers

At the time of writing this book, the available version of the wlp cookbook implements functions to start and stop Liberty profile servers only as resource modules and not as recipes. Because only recipes can be specified in a node configuration's run-list, we must create our own recipes that use these resources.

When updating the wlp cookbook sample, we do not create our recipes in the existing wlp cookbook but create a new cookbook named todoapp that will hold our recipes. See Example 6-18 on page 79.

Example 6-18 Creating a new cookbook

```
chef@chefmaster:~/mychefrepo$ knife cookbook create todoapp -o cookbooks
** Creating cookbook todoapp
** Creating README for cookbook: todoapp
** Creating CHANGELOG for cookbook: todoapp
** Creating metadata for cookbook: todoapp
```

Because we want to use resources defined in the wlp cookbook in our recipes part of the todoapp cookbook, we must declare a dependency to the wlp cookbook. This is done by adding the following line to the `mychefrepo/cookbooks/todoapp/metadata.rb` file (Example 6-19):

```
depends 'wlp'
```

Example 6-19 Cookbook metadata file with declared wlp dependency

```
chef@chefmaster:~/mychefrepo$ cat cookbooks/todoapp/metadata.rb
name             'todoapp'
maintainer       'YOUR_COMPANY_NAME'
maintainer_email 'YOUR_EMAIL'
license          'All rights reserved'
description      'Installs/Configures todoapp'
long_description IO.read(File.join(File.dirname(__FILE__), 'README.md'))
version          '0.1.0'
depends          'wlp'
```

After making the resources of the wlp cookbook available to the local name space of recipes in our cookbook, we now create a recipe named *start-servers* that will start all servers defined in the node's attributes map under the `["wlp"]["servers"]` key that have the `enabled` attribute set to `TRUE`. That is, we simply reuse the `wlp::serverconfig` recipe's attribute configuration in the node configuration file.

In mychefrepo, create a new recipe at `cookbooks/todoapp/recipes/start-servers.rb` with the content shown in Example 6-20.

Example 6-20 start-servers recipe

```
#
# Cookbook Name:: todoapp
# Recipe:: start-servers

# iterate over all entries under the ["wlp"]["servers"] key
# of the current node's attributes map
node[:wlp][:servers].each_pair do |key, value|
   map = value.to_hash()
   enabled = map.fetch("enabled", nil)

   # only consider enabled servers
   if enabled.nil? || enabled == true
      # determine the server-name based on the "serverName" attribute value
      # if exists, otherwise use the key of the current "servers" map entry
      serverName = map.fetch("serverName", nil) || key

      # for each server execute the recipe wlp_server::start
      wlp_server "#{serverName}" do
         # clean all caches before server start
         clean true
         action :start
      end

   end
end
```

To use the start-servers recipe, we add it to the run-list of the node configuration. Therefore alter the run-list declaration in the **nodes/chefslave.json** file as follows:

```
"run_list":["recipe[wlp::default]", "recipe[wlp::serverconfig]",
"recipe[todoapp::start-servers]"]
```

Rerunning the deployment using the **knife solo cook** command now starts the servers defined in the node configuration. See Example 6-21.

Example 6-21 Executing a deployment that starts Liberty profile servers

```
chef@chefmaster:~/mychefrepo$ knife solo cook chef@chefslave
Running Chef on chefslave...
Checking Chef version...
Uploading the kitchen...
Generating solo config...
Running Chef...
Starting Chef Client, version 11.8.0
Compiling Cookbooks...
Converging 22 resources
Recipe: wlp::default
   ... (output omitted)
Recipe: todoapp::start-servers
   ... (output omitted)
Recipe: <Dynamically Defined Resource>
   * service[wlp-todo-ui-rest] action nothing (skipped due to action :nothing)
   * template[/etc/init.d/wlp-todo-ui-rest] action create
      - create new file /etc/init.d/wlp-todo-ui-rest
      - update content in file /etc/init.d/wlp-todo-ui-rest from none to dea6b2
      - change mode from '' to '0755'
      - change owner from '' to 'root'
      - change group from '' to 'root'
   * service[wlp-todo-ui-rest] action enable
      - enable service service[wlp-todo-ui-rest]
   * service[wlp-todo-ui-rest] action start
      - start service service[wlp-todo-ui-rest]

Recipe: todoapp::start-servers
   ... (output omitted)
Recipe: <Dynamically Defined Resource>
   * service[wlp-todo-ui-wicket] action nothing (skipped due to action :nothing)
   * template[/etc/init.d/wlp-todo-ui-wicket] action create
      - create new file /etc/init.d/wlp-todo-ui-wicket
      - update content in file /etc/init.d/wlp-todo-ui-wicket from none to 00b426
      - change mode from '' to '0755'
      - change owner from '' to 'root'
      - change group from '' to 'root'
   * service[wlp-todo-ui-wicket] action enable
      - enable service service[wlp-todo-ui-wicket]
   * service[wlp-todo-ui-wicket] action start
      - start service service[wlp-todo-ui-wicket]
```

You can verify that the servers are running now by checking for the existence of the corresponding Java process on the slave node and also by using the service scripts that were conveniently created by the `wlp::start` resource. See Example 6-22.

Example 6-22 Executing a deployment that starts Liberty profile servers

```
chef@chefslave:~$ ps -efww | grep java | grep -v grep

wlp      18211     1  0 23:02 ?        00:00:04
/usr/lib/jvm/java-6-openjdk-i386/jre/bin/java -XX:MaxPermSize=256m
-javaagent:/opt/ibm/wlp/bin/tools/ws-javaagent.jar -jar
/opt/ibm/wlp/bin/tools/ws-server.jar todo-ui-rest --clean

wlp      18314     1  0 23:02 ?        00:00:04
/usr/lib/jvm/java-6-openjdk-i386/jre/bin/java -XX:MaxPermSize=256m
-javaagent:/opt/ibm/wlp/bin/tools/ws-javaagent.jar -jar
/opt/ibm/wlp/bin/tools/ws-server.jar todo-ui-wicket --clean

chef@chefslave:~$ sudo /etc/init.d/wlp-todo-ui-rest status
Server todo-ui-rest is running with process ID 18211.

chef@chefslave:~$ sudo /etc/init.d/wlp-todo-ui-wicket status
Server todo-ui-wicket is running with process ID 18314.
```

7

Working with third-party tools on Liberty profile server

This chapter demonstrates how to use third-party tools, such as Apache ActiveMQ, for implementing JMS functionality using Liberty profile. It also demonstrates how to integrate third-party mail servers into Liberty profile for sending and receiving mail from your application code.

7.1 Apache ActiveMQ with Liberty profile server

Apache ActiveMQ is one of the most popular open source message-oriented middleware products in existence today. It is a prime product used in enterprise environments because it supports many advanced features such as multiple instances to store messages, and clustering environments.

Apache ActiveMQ is fast and offers several important features:

- Supports many cross language clients and protocols.
- Has easy-to-use Enterprise Integration Patterns.
- Supports advanced features such as message groups, virtual destinations, wildcards and composite destinations.
- Fully supports JMS 1.1 and J2EE 1.4 with support for transient, persistent, transactional and XA messaging.
- Supports pluggable transport protocols such as in-VM, TCP, SSL, NIO, UDP, multicast, JGroups and JXTA transports.
- Supports fast persistence using JDBC along with a high performance journal
- Is designed for high performance clustering, client-server, and peer based communication.
- Has REST API to provide technology-independent and language-neutral web-based API to messaging.
- Uses Ajax to support web streaming to web browsers using pure DHTML, allowing web browsers to be part of the messaging fabric.
- CXF and Axis Support so that ActiveMQ can be easily dropped into either of these web service stacks to provide reliable messaging.

Liberty profile server supports the following JMS messaging providers:

- Liberty profile server embedded messaging engine, as the JMS messaging provider
- Service integration bus, which is the default messaging provider of WebSphere Application Server
- WebSphere MQ messaging provider, which uses the WebSphere MQ system as the provider

In addition, external JMS providers such as Apache ActiveMQ can also be integrated with Liberty profile server as mentioned in the next example.

7.1.1 Example

In this example, you write, deploy, and test a simple JMS application on Liberty profile server. The application sends and consumes a message through an ActiveMQ instance running outside the Liberty profile server.

Prerequisites

Complete these prerequisite steps for writing and testing the JMS application on Liberty profile server:

1. Download Apache ActiveMQ for Windows from the following location:

   `http://www.apache.org/dyn/closer.cgi?path=/activemq/apache-activemq/5.8.0/apache-activemq-5.8.0-bin.zip`

2. Extract the ZIP file and start ActiveMQ by issuing the **`activemq`** command from the `bin` folder.

   On a successful start, ActiveMQ starts listening for connection at `tcp://localhost:61616`

3. You can also open ActiveMQ Web Console using the following URL:

   `http://localhost:8161/admin/`

4. The default user name and password for ActiveMQ Web Console login is `admin`.

Writing, deploying, and testing the JMS sample application

Complete the following steps to write a simple JMS application that can send and receive a message using JNDI support in ActiveMQ:

1. Create a simple web application using Eclipse or Rational Application Developer.
2. Add the `jndi.properties` file (in Figure 7-1) to the class path, which is the `/src` folder.

```
java.naming.factory.initial =
org.apache.activemq.jndi.ActiveMQInitialContextFactory

# use the following property to configure the default connector
java.naming.provider.url = tcp://localhost:61616

# use the following property to specify the JNDI name the connection factory
# should appear as.
#connectionFactoryNames = connectionFactory, queueConnectionFactory,
topicConnectionFactry
connectionFactoryNames = connectionFactory

# register some queues in JNDI using the form
# queue.[jndiName] = [physicalName]
queue.MyQueue = Q1
```

Figure 7-1 jndi.properties

3. Copy the JMS implementation classes (`activemq-all.jar`) from the `lib` folder of ActiveMQ installation directory to the `lib` folder of the web application.

Bundled: The JMS implementation classes (`activemq-all.jar`) are bundled within the application because Liberty profile server does not provide any Apache ActiveMQ implementation.

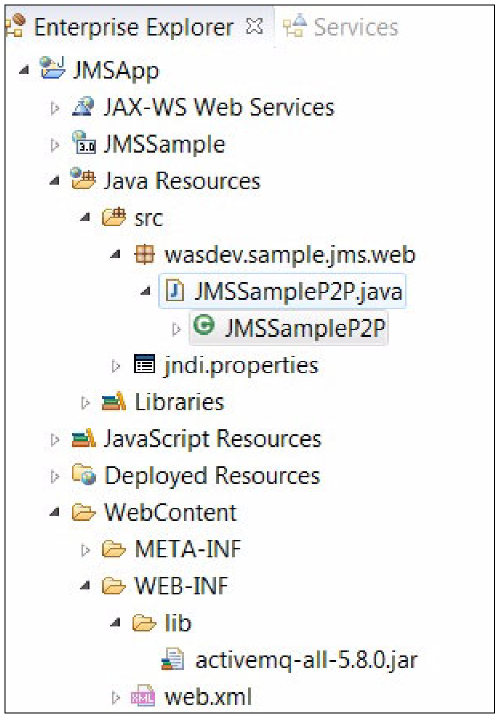
4. Write a simple servlet that gets InitialContext and then the resource `JNDI names` by reading the `jndi.properties` file (in Figure 7-1).

Using the resource JNDI names and the JMS implementation classes, the servlet sends and receives a message as shown in Figure 7-2.

```
public void sendAndReceive(HttpServletRequest request,
        HttpServletResponse response) throws Exception {

      PrintWriter out = response.getWriter();
      out.println("SendAndReceive Started");

      Context ctx = new InitialContext();

      QueueConnectionFactory cf1 = (QueueConnectionFactory)
ctx.lookup("connectionFactory");
      Queue queue = (Queue) ctx.lookup("MyQueue");
      out.println("QCF and Queue lookup completed !!");

      QueueConnection con = cf1.createQueueConnection();

      // start the connection to receive message
      con.start();

      // create a queue session to send a message
      QueueSession sessionSender = con.createQueueSession(false,
            javax.jms.Session.AUTO_ACKNOWLEDGE);

      QueueSender send = sessionSender.createSender(queue);
      out.println("Message sent successfully");
      // send a sample message
      send.send(sessionSender.createTextMessage("Liberty Sample Message"));

      // create a queue receiver object
      QueueReceiver rec = sessionSender.createReceiver(queue);

      // receive message from Queue
      TextMessage msg = (TextMessage) rec.receive();

      out.println("Received Message Successfully :" + msg);

      if (con != null)
         con.close();
      out.println("SendAndReceive Completed");
   }// end of SendAndReceive
```

Figure 7-2 sendAndReceive Method inside servlet

The project hierarchy now looks like Figure 7-3.


Enterprise Explorer ☒ Services

- JMSApp
  - JAX-WS Web Services
  - JMSSample
  - Java Resources
    - src
      - wasdev.sample.jms.web
        - JMSSampleP2P.java
          - JMSSampleP2P
      - jndi.properties
    - Libraries
  - JavaScript Resources
  - Deployed Resources
  - WebContent
    - META-INF
    - WEB-INF
      - lib
        - activemq-all-5.8.0.jar
      - web.xml

Figure 7-3 JMSApp Project Hierarchy

5. Save and export the application as `JMSApp.war` file.
6. Add servlet-3.0 to the Liberty `server.xml` file as shown in Figure 7-4.

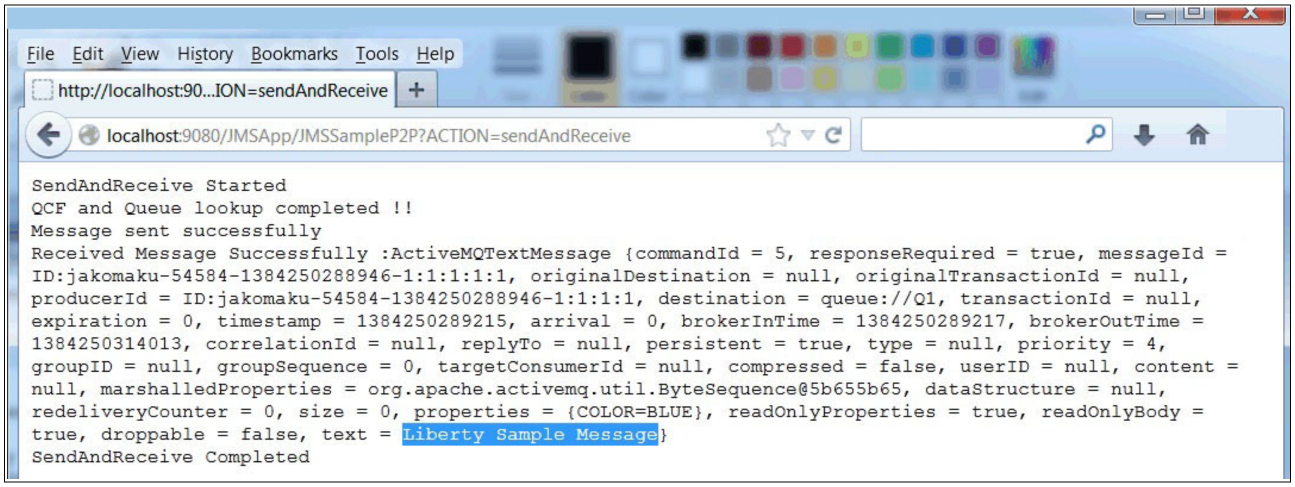
```
<featureManager>
       <feature>servlet-3.0</feature>
</featureManager>
```

Figure 7-4 Liberty server.xml

7. Deploy the `JMSApp.war` file on the Liberty profile server by copying it to the `dropins` folder
8. Point to the following URL for invoking the JMS Servlet on Liberty profile server:

   `http://<hostname>:<httpport>/JMSApp/JMSSampleP2P?ACTION=sendAndReceive`

   The servlet should send and receive a message using the Apache ActiveMQ JMS provider as shown in Figure 7-4.
9. You can also view the message being sent and consumed on the Apache ActiveMQ Web Console. See Figure 7-5 on page 88.


File Edit View History Bookmarks Tools Help
http://localhost:90...ION=sendAndReceive
localhost:9080/JMSApp/JMSSampleP2P?ACTION=sendAndReceive

```
SendAndReceive Started
QCF and Queue lookup completed !!
Message sent successfully
Received Message Successfully :ActiveMQTextMessage {commandId = 5, responseRequired = true, messageId =
ID:jakomaku-54584-1384250288946-1:1:1:1:1, originalDestination = null, originalTransactionId = null,
producerId = ID:jakomaku-54584-1384250288946-1:1:1:1, destination = queue://Q1, transactionId = null,
expiration = 0, timestamp = 1384250289215, arrival = 0, brokerInTime = 1384250289217, brokerOutTime =
1384250314013, correlationId = null, replyTo = null, persistent = true, type = null, priority = 4,
groupID = null, groupSequence = 0, targetConsumerId = null, compressed = false, userID = null, content =
null, marshalledProperties = org.apache.activemq.util.ByteSequence@5b655b65, dataStructure = null,
redeliveryCounter = 0, size = 0, properties = {COLOR=BLUE}, readOnlyProperties = true, readOnlyBody =
true, droppable = false, text = Liberty Sample Message}
SendAndReceive Completed
```

Figure 7-5 SendAndReceive web page

7.2 Apache James with Liberty profile server

Apache James is a 100% pure Java SMTP, POP3 Mail server, IMAP (James V3) and NNTP News (James V2) server designed to be a complete and portable enterprise mail/messaging engine solution based on currently available open messaging protocols.

Among the benefits of James over other mail platforms is its support for building custom mail handling applications.

Liberty profile server does not include any JavaMail features. However one can easily write and deploy JavaMail applications on Liberty profile server by providing the JavaMail implementation within the application as shown in the next example.

7.2.1 Example

In this example, you write, deploy, and test a simple JavaMail application on the Liberty profile server. This application sends mail to a valid recipient through an externally running Apache James Mail server instance.

Prerequisites

The following prerequisites must be met before writing and testing the JavaMail application on Liberty profile server:

1. Download Apache James from the following location:

   `http://www.motorlogy.com/apache//james/server/apache-james-2.3.2.zip`

2. Extract the ZIP file and start James Mail server by issuing the **`run.bat`** command from inside the `bin` folder.
3. Look for the `Apache James is successfully started in XXXX milliseconds` message in the command-line console.

Writing, testing, and deploying the Java Mail sample application

Use the following steps to write a simple JavaMail application that can send a mail to a valid recipient through James Mail server:

1. Create a simple web application using Eclipse or Rational Application Developer.
2. Copy the JavaMail implementation classes (`mail.jar`) from the `lib` folder of James Mail Server installation directory to the `lib` folder of the web application.

   Bundled: The JavaMail implementation classes (`mail.jar`) are bundled within the application because the Liberty profile server does not provide any Apache JavaMail implementation.

3. Add a new JSP page with the content from Figure 7-6. Make sure to use the correct “to” and “from” values.

```
<%@ page import="java.util.*, javax.mail.*, javax.mail.internet.*" %>
<%
String host = "localhost";
String user = "xxxx@gmail.com";
String pass = "XXXX";
String to = "xxxx@in.ibm.com";
String from = "xxxxx@gmail.com";
String subject = "WELCOME";
String messageText = "Test Mail !! Please ignore...";
boolean sessionDebug = false;
Properties props = System.getProperties();
props.put("mail.host", host);
props.put("mail.transport.protocol", "smtp");
props.put("mail.smtp.auth", "true");
Session mailSession = Session.getDefaultInstance(props, null);
mailSession.setDebug(sessionDebug);
Message msg = new MimeMessage(mailSession);
msg.setFrom(new InternetAddress(from));
InternetAddress[] address = {new InternetAddress(to)};
msg.setRecipients(Message.RecipientType.TO, address);
msg.setSubject(subject);
msg.setSentDate(new Date());
msg.setText(messageText);
Transport transport = mailSession.getTransport("smtp");
transport.connect(host, user, pass);
transport.sendMessage(msg, msg.getAllRecipients());
out.println("Sent Message");
transport.close();
%>
```

Figure 7-6 WebContent/mail.jsp

4. Save and export the application as a `JavaMail.war` file.
5. Add `jsp2.2` feature to the Liberty `server.xml` file as shown in Figure 7-7.

```
<featureManager>
        <feature>jsp-2.2</feature>
   </featureManager>
```

Figure 7-7 Liberty server.xml

6. Deploy the `JavaMail.war` on the Liberty profile server by copying it to the `dropins` folder.
7. Use the following URL to invoke the JSP Page on Liberty profile server:

   `http://<hostname>:<httpport>/javamail/mail.jsp`

A message should now be sent to the recipient mentioned in the JSP page using the James mail server that is running outside of Liberty profile server.

A

Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

`ftp://www.redbooks.ibm.com/redbooks/SG248194`

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG24-9194.

Using the Web material

The additional Web material that accompanies this book includes the following files:

File name	*Description*
SG248194.zip	Todo Sample Source code

Downloading and extracting the Web material

Create a subdirectory (folder) on your workstation, and extract the contents of the web material `.zip` file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *WebSphere Application Server Liberty Profile Guide for Developers*, SG24-8076
- *WebSphere Application Server V8.5 Administration and Configuration Guide for Liberty Profile*, SG24-8170
- *WebSphere Application Server V8.5 Administration and Configuration Guide for the Full Profile*, SG24-8056-01

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- Liberty profile: Configuration elements in the server.xml file:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.nd.doc%2Fautodita%2Frwlp_metatype_4ic.html
- Liberty profile: Security:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.nd.doc%2Fae%2Fcwlp_sec.html
- Configuration elements in the server.xml file:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.core.doc%2Fautodita%2Frwlp_metatype_core.html
- Tuning the Liberty profile:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.core.doc%2Fae%2Ftwlp_tun.html
- Liberty features:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/topic/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/rwlp_feat.html
- Liberty profile V8.5.5.Next Alpha:

  https://www.ibmdw.net/wasdev/

- Packaging a Liberty profile server from the command prompt:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/index.jsp?topic=%2Fcom.ibm.websphere.wlp.core.doc%2Fae%2Ftwlp_setup_package_server.html

- Programming model support:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/topic/com.ibm.websphere.wlp.nd.doc/ae/rwlp_prog_model_support.html

- Developing applications with WebSphere Developer Tools and Liberty profile:

  https://www.ibmdw.net/wasdev/docs/developing-applications-wdt-liberty-profile/

- Apache Maven installation instructions:

  http://maven.apache.org/download.cgi#Installation_Instructions

- Maven Integration (m2e):

  http://www.eclipse.org/m2e/

- Subclipse download and install:

  http://subclipse.tigris.org/servlets/ProjectProcess?pageID=p4wYuA

- Orika¶:

  https://code.google.com/p/orika/

- Arquillian:

  http://arquillian.org/

- WLP V8.5 - Managed:

  https://docs.jboss.org/author/display/ARQ/WLP+V8.5+-+Managed

- flapdoodle-oss / de.flapdoodle.embed.mongo:

  https://github.com/flapdoodle-oss/de.flapdoodle.embed.mongo

- Spring Data – One API To Rule Them All:

  http://www.infoq.com/articles/spring-data-intro

- H2 Database Engine:

  http://www.h2database.com/

- Getting Started with dojox/mobile:

  http://dojotoolkit.org/documentation/tutorials/1.9/mobile/tweetview/getting_started/

- JWeUnit:

  http://jwebunit.sourceforge.net/

- REST-assured:

  https://code.google.com/p/rest-assured/

- An Overview of Chef:

  http://docs.opscode.com/chef_overview.html

- Opscode Chef Glossary:

  https://wiki.opscode.com/display/chef/Glossary

- knife-solo:

  http://matschaffer.github.io/knife-solo/

- DevOpsCasts Chef Solo Basics:

  http://devops.mashion.net/2011/08/04/chef-solo-basics/
- Ubuntu Server:

  http://www.ubuntu.com/download/server
- The Apache Software Foundation - Apache Download Mirrors:

  http://www.apache.org/dyn/closer.cgi?path=/activemq/apache-activemq/5.8.0/apache-activemq-5.8.0-bin.zip
- Download Apache James:

  http://www.motorlogy.com/apache//james/server/apache-james-2.3.2.zip
- Downloads WebSphere Application Server Liberty Profile:

  http://wasdev.net/downloads

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

IBM Redbooks

Configuring and Deploying Open Source with WebSphere Application Server Liberty Profile

(0.2”spine)
0.17”<->0.473”
90<->249 pages

Configuring and Deploying Open Source with WebSphere Application Server Liberty Profile

Learn about Liberty profile to develop and and test your web and OSGi applications

Install and configure several open source technologies with Liberty profile

Deploy Liberty with Chef framework

This IBM Redbooks publication explains the capabilities of IBM WebSphere Application Server Liberty profile, which is extremely lightweight, easy to install, and fast to use. Liberty profile provides a convenient and capable platform for developing and testing your web and OSGi applications. The Liberty profile server is built using OSGi technology and concepts. The fit-for-purpose nature of the run time relies on the dynamic behavior inherent in the OSGi framework and service registry. As bundles are installed or uninstalled from the framework, their services are automatically added or removed from the service registry. The result is a dynamic, composable run time that can be provisioned with only what your application requires and responds dynamically to configuration changes as your application evolves.

This book can help you install, customize, and configure several popular open source technologies that can be deployed effectively with the WebSphere Application Server Liberty profile.

Popular open source toolkits for the Liberty profile server were selected for this book based on significant enhancements they provide to the web application development process.

In this book, a “Todo” sample demonstrates the use of multiple open source frameworks or toolkits with the Liberty profile server. The Todo sample is a simple application that can be used to create, update, and delete to-do items and to-do lists, and put the to-do items into a related to-do list.

SG24-8194-00 ISBN 0738439258