Application Developer

LIFERAY Training

2.1 Developing in a Module Framework

2.1.1 The Liferay Platform

Takeaway

- The Liferay Platform provides a stable dev environment.
- Development is based on highly modular components.
- Modules are developed independently.

The Liferay Platform provides a reliable, stable environment to develop in. New features and customizations are created within a modular, component-based framework. Instead of relying on a large, monolithic codebase, developers are able to leverage smaller modules to do what they need. Modules are developed independent of one another, allowing for app development as simple or complex as you want.

2.1.2 The Modular Framework

Takeaway

- Liferay's Module Framework provides component architecture
- Apps, services, and other functionality are *modules*
- Modules contain *components* that are building blocks

Liferay's Module Framework is the heart and soul of developing in the platform. New development, whether it's apps, services, or any collection of files, starts with creating *modules*. A module is a self-contained, packaged, set of functionality. Modules can contain *components*, which are the building blocks of features and interaction in Liferay. Instead of needing to create specialized plugins that are tightly coupled to a version of the platform, *modules* and *components* can operate anywhere the code can run.

2.1.3 Monolithic or Modular

Takeaway

- Monolithic architecture:

  - Tightly coupled
  - Inflexible
  - Static dependencies

- Modular, component architecture:

  - Loose coupling
  - Flexible, easy to change
  - Dynamic dependencies

Modular development is an architecture that is somewhat more complex than a straightforward, monolithic-style system. However, the flexibility modular architecture gives both the developer and the end user makes long-term maintenance and upgrading easier. With the modules loosely coupled to their dependencies, it's easier to move modules in and out of any given environment. As long as dependencies are satisfied, the module can be deployed successfully. Additionally, within a component–based architecture, it is easy to integrate many external services and other dependencies without too much fuss. In a monolithic system, modifying dependencies may be complex, and can usually affect other applications in the system.

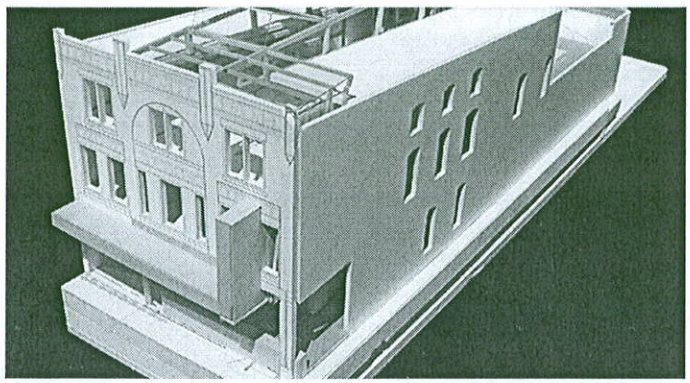
2.1.4 A Big Example

Takeaway

(Figure 2.1, page 36)

Let's take this abstract concept to a slightly different context. If I wanted to build a scale model of a castle, a house, or Liferay headquarters, I could do it rather quickly and easily in a monolithic way. I could cut and shape the walls to the exact dimensions I need for that building. I could make sure the chimney was made to fit the exact roof I also made by hand. I could paint each piece the color I need, knowing that I have full control over each aspect. However, if I wanted to change — *anything* — on the model, it could require a lot of in depth work. Modifying the walls would require additional engineering, probably equal to the initial effort. Some things might be standard — gutters, window panes, doors — but very few could be swapped out easily. If I want to make another scale model of a different building, I'd have to construct it from scratch: my previous model–building would have little effect on my new model, other than experience.

Figure 2.1:

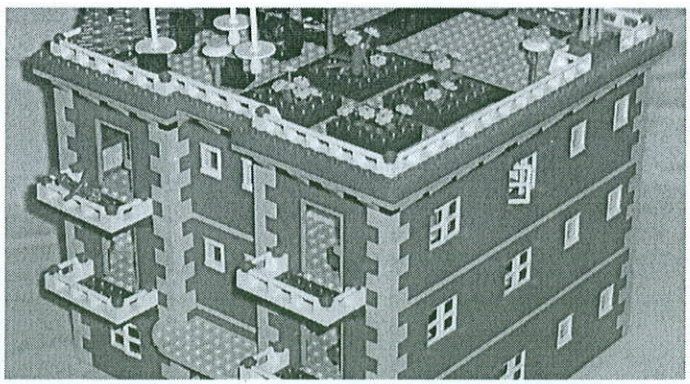


2.1.5 A Flexible Solution

Takeaway

(Figure 2.2, page 36)

Figure 2.2:



Enter components. If I instead build my model building from small components, all independent of the others, but complete in and of themselves, with discrete functionality, I can put together any structure — whether simple or complex — easily. Any changes I want to make can be done more simply by replacing some of the components, or adding new ones. Maintaining this model over time will be easier, since all components meet the same basic rules: they all fit together.

2.1.6 A Historical Example

Takeaway

A more salient example can be taken from how Operating Systems treat access to system resources. This is handled through the kernel. In an OS, the kernel can be a monolithic kernel or a microkernel. Here are the differences:

- Monolithic Kernel:
  - Controls the system, from low–level hardware to user-level services.
  - Essential features and services are incorporated into the system code.
  - If anything breaks, gets moved, or an upgrade happens, applications can be broken, occasionally causing the system to shut down.
- Microkernel:
  - The Microkernel only contains the barebones to keep the system up.
  - Essential features (file access, graphics card access, input and output drivers, etc.) are handled by components.
  - Components sit on top of the core platform and provide essential functions to other applications.
  - If something breaks, it's often possible to simply replace or restart a component.

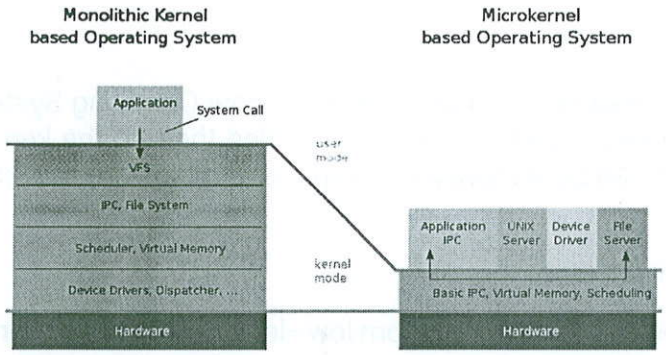
(Figure 2.3, page 38)

2.1.7 Moving into the Module Framework

Takeaway

- Developing modules means:
  - Creating small modules
  - Publishing and consuming other modules' features
  - Working without static dependencies

Figure 2.3:



- To use the Module Framework, you need:
  - Modules
  - Components
  - Application Lifecycle

Moving into a world of mostly, or even exclusively, modular development may take some getting used to. It certainly changes some of the ways we approach solving problems. We will develop applications as multiple smaller modules, rather than one, monolithic application. This will also make upgrades and changes easy to incorporate. In order to successfully use the Module Framework, we need to understand *modules*, *components* and the *application lifecycle*.

2.2 Development Tools

2.2.1 Development Tools

- A wide variety of tools can be used to write apps:
  - Standard Java EE development environment
  - Java IDE like Eclipse or IntelliJ
  - Bnd and Bndtools
- Additional tools can be used to simpify development further, such as:
  - Maven

  - Ant + Ivy
  - Gradle
- We'll want to use some more substantial tools to make developing this big app easier.

2.2.2 Liferay's Development Environment

- When setting up our basic development environment, we set up:
  - Liferay IDE with Eclipse
    * `https://www.liferay.com/downloads/liferay-projects/liferay-ide`
  - a Liferay Workspace with a Liferay 7 server
- These are part of the full development environment we'll take advantage of.
- Liferay IDE plugins provide a number of enhancements and shortcuts to make it easy to build and install on Liferay.
- Liferay Workspace provides a place for us to manage large projects and build easily.
- What does the full suite of tools look like?

2.2.3 The Build Tools - Gradle

- Creating larger, more complex projects means we need to have a build management system in place.
- We can already use Bnd for handling the build of modules easier, but it's still not streamlined for Liferay.
- *Gradle* is a build and dependency management system for Java.
- It is both simple and flexible, making it a powerful tool for us to use.
- Gradle projects have a default project layout, which we can take advantage of to simplify our build files.
- Plugins can be applied to Gradle projects to do additional processing – for instance, we can use a plugin to leverage Bnd in a Gradle build.

- This fits in perfectly with a larger development environment.
- You can find more information on Gradle on the project's website:
  - *http://gradle.org/*

2.2.4 Liferay Blade

- Developing modules involves reusing a lot of the same configuration.
- To make it easier, as well as providing templates for different apps and modules, Liferay created a collection of skeleton projects.
- These projects make up BLADE (Bootstrap Liferay Advanced Development Environments).
  - BLADE can be found on Github: *https://github.com/rotty3000/blade*
- The BLADE projects provide an easy way to get started on a new module or set of modules: copy and paste, then modify the details and add your own code.
- BLADE projects demonstrate development on Liferay 7, and can be used in any IDE you wish.
- Projects come in different flavors, whether you want to use Bnd, Bnd and Gradle, or the full set with Bnd, Gradle and the Liferay Gradle plugins.

2.2.5 Liferay Blade CLI

- BLADE is incredibly useful for creating modules, but could be streamlined a bit.
- Liferay also provides an interactive, automated interface into creating projects from BLADE.
- *blade CLI* is a project compatible with JPM4J that provides:
  - Access to the BLADE projects from the command line
  - Creation commands to replace placeholder values in the templates

  - Commands for building and installing modules on a Liferay instance

- More information on *blade CLI*, including instructions to install it, can be found at the Github project:
  - *https://github.com/gamerson/liferay-blade-tools*

2.2.6 Liferay Workspaces

- BLADE and blade CLI provide great templates for projects, and project stubs.
- It's not a full dev solution, but fits into existing structures well.
- A completely configured development environment needs more than just some project templates and build tools.
- *Liferay Workspace* is a simple development environment built on BLADE, blade CLI and Gradle.
- Liferay Workspace provides a project structure to follow that's simple, and can be used as the base of a source code repository.
- In addition to the project structure and tools, Liferay Workspace provides automation for executing Gradle, installing and using a Liferay bundle, and build large groups of projects.
- Liferay IDE provides a front end in Eclipse for Liferay Workspace, including blade CLI.

2.3 What are Modules?

2.3.1 What is a Module?

Takeaways

- A *module* is a self-contained unit to deploy in Liferay.
- A *module* contains:
  - Java classes
  - A manifest file

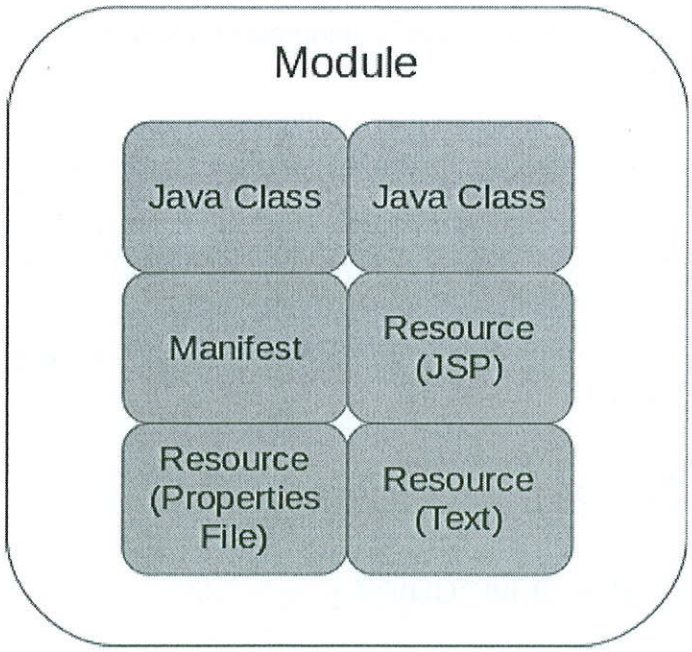
  - Resources:
    * JSPs
    * Properties files
    * Binary or text data

- Modules are packaged as JAR files.

The *module* is the only plugin type in the Liferay 7 Platform, and is the basic unit of development. Modules may contain any combination of **Java classes, JSPs, properties files, images,** or any other binary and text data your application needs. Once a module is created, it is packaged as a basic JAR file for deployment. That's it — no special container or processing needed.

(Figure 2.4, page 42)

Figure 2.4:



2.3.2 Modules, Servlets, and Portlets — Oh My!

Takeaways

- Modules make up all applications:

- Shell commands
- Services
- Servlets
- Portlets

If you have prior experience with Java EE development technologies, such as servlets and portlets, these applications are also deployed as *modules*. Whether we're writing services, shell commands, portlets or servlets, the module is the basic deployable container.

2.3.3 The Essentials of a Module

Takeaways

- In order for a module to be valid, the JAR file needs the following:
  - Java classes
  - A valid manifest (`MANIFEST.MF`)

Resources are not necessary for a module to be valid, but are used by your application.

(Figure 2.5, page 44)

2.3.4 A Module's Name

Takeaways

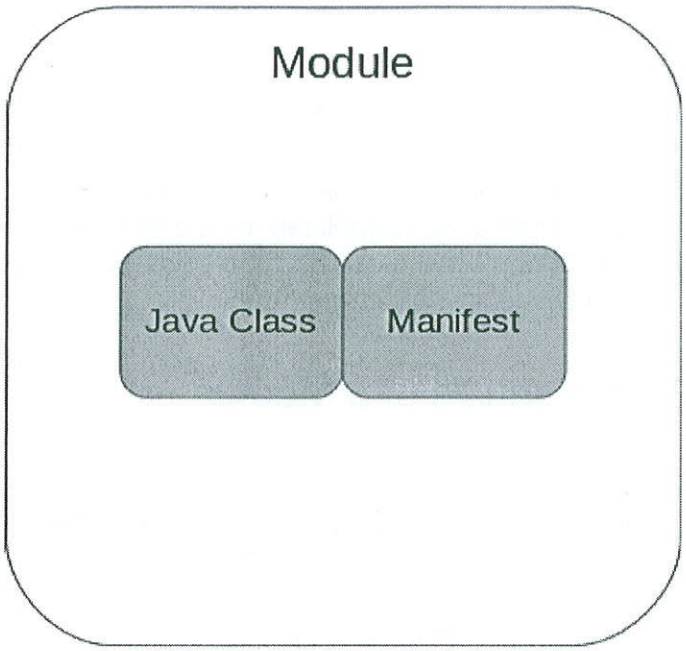
- Along with Java classes and a valid manifest, modules must have a *unique identifier*.
- The unique name of a module is composed of:

  ```
  Module Name + Module Version
  ```

- We declare the module name and module version in the module's manifest.

When deploying modules, Liferay needs to uniquely identify your module so that upgrading and redeploys can go smoothly. Each module is uniquely identified by its *symbolic name* and *version number*. So for a module whose name is `com.liferay.training.test` and a version number of

Figure 2.5:



`1.0.0`, it will be uniquely identified as `com.liferay.training.test,1.0.0`. If I then deploy a new version of the same symbolic name, say, version `1.0.1`, that bundle will be identified as `com.liferay.training.test,1.0.1`. This means that you can deploy different versions of the same module concurrently to satisfy the requirements of different applications.

2.3.5 Module Review

Takeaways

- *Modules* are the basic plugin type in Liferay.
- A module may contain any combination of *code* and *resources*
- Modules are packaged as *JAR* files.
- Modules are identified by a unique *name* and *version number*.
- Multiple versions of the *same module* can be deployed concurrently.

2.4 What are Components?

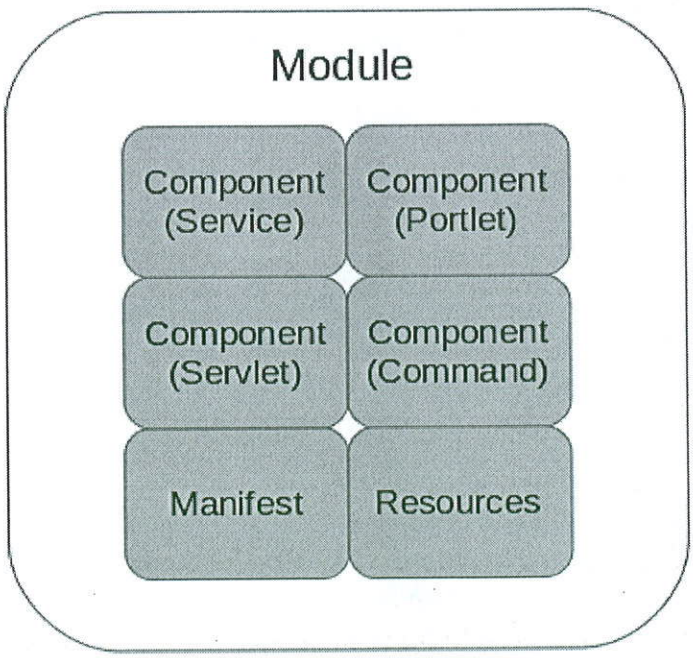
2.4.1 What is a Component?

Takeaways

- A *component* is an object that provides a feature or functionality:
  - Service
  - Servlet
  - Portlet
  - Shell command
- Like a building block, components can be used together for large applications.
- Components are contained within *modules*.

(Figure 2.6, page 45)

Figure 2.6:



All applications need business logic, lifecycle management, and occasionally services and other features. We can break down these various functional pieces as individual *components*. Each component provides a specific function or feature that we need, either in an application, or as a standalone service or command. Components are simply Java objects that implement functionality — service, portlet, etc. — and live inside modules.

2.4.2 Why Components?

Takeaways

- Components are:
  - Reusable and readily available
  - Building blocks for an application

Ultimately, *components* are meant to help enforce the idea of modularity by being reusable and readily available. Components with the desired features are snapped together, like building blocks, to create an application. Components are not bound to a specific application and can be reused throughout any number of applications.

2.4.3 Building A Simple Component

Takeaways

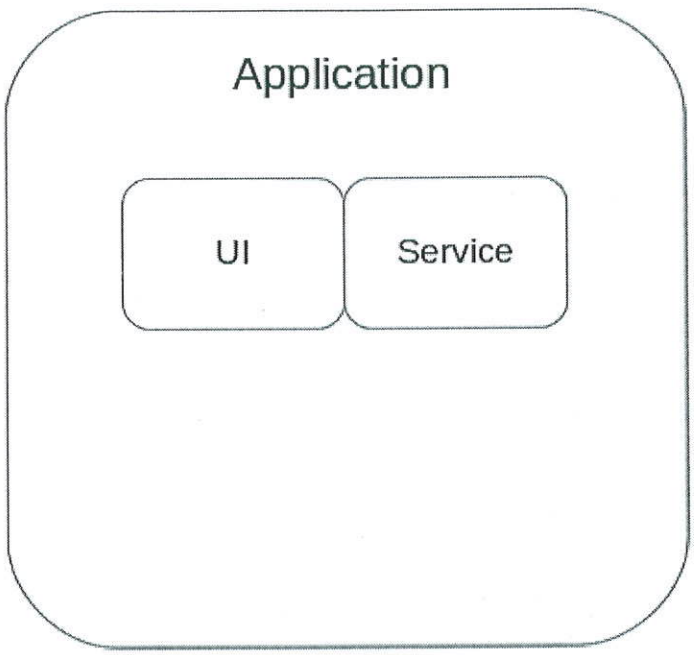
- Suppose we want to build out a simple application that has the following features:
  - A very simple UI to view data
  - A service that retrieves data to be displayed
  - An object that handles the application logic
- How could this be developed in the Module Framework?

A simple example should highlight what an application might look like in Liferay 7. If we wanted to create an application that displayed data from a data source in a nice UI in Liferay, what would we need to implement? We'd need to design and create the UI for displaying the data, a service

for retrieving the data from said data source, and an object for handling application logic, such as displaying the UI, retrieving data from the service, and handing data down to the UI. What would this look like if we developed it in the Module Framework?

(Figure 2.7, page 47)

Figure 2.7:



2.4.4 Sample Components

Takeaways

- The simple UI could be implemented as JSPs.
- The service could be implemented as a set of Java objects, defining the service API and providing an implementation.
- The application object could be implemented as a Java portlet.
- How does each piece fit into components and modules?

Our functionality can be implemented in a pretty straightforward way — the UI can be done through JSPs, the service implemented as Java objects

that define the service API and implementation, and the application logic can be controlled as a Java portlet. But how do we create each of these functional pieces in Liferay and package it all together?

2.4.5 A Component-based Application

Takeaways

- Use a *module*:
  - JSPs are *resources*.
  - A service is a *component*.
  - A portlet (application) is a *component*.
- Use two *modules*:
  - An application module with:
    * JSPs as *resources*
    * Portlet as a *component*
  - A Service module with:
    * Service as a *component*

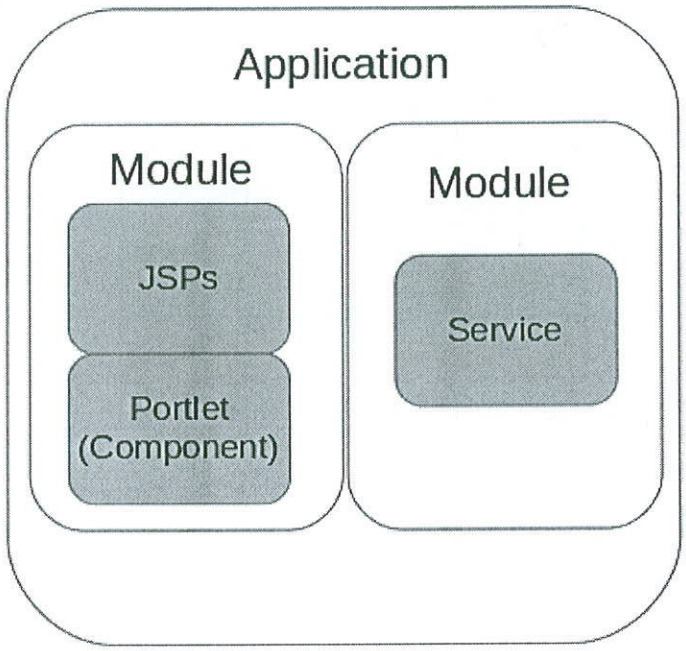
In the Module Framework, we can package this application as one or more *modules*. Within the modules, which are just containers for code and resources, we can implement the application as JSPs stored as *resources*, the portlet as a *component*, and the service as a *component*. We could also split the service off into its own module, so other applications can use it easily. Simply have the application in one module and the service in the other. Within Liferay's Module Framework, applications can be flexible, atomic, and easy to change and upgrade.

(Figure 2.8, page 49)

2.4.6 Putting It All Together

- Modules are the basic deployable container used in Liferay platform.
- Within the module is the component.
- The component provides the implementation of a feature for an application, such as business logic, lifecycle management, and sometimes services.

Figure 2.8:



- We can then build out applications, using various modules containing components, that provide the desired features and services.

2.5 The Application Lifecycle

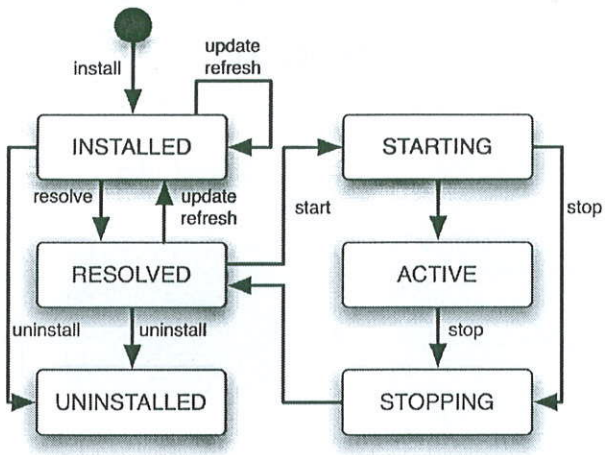
2.5.1 Deploying Applications

Takeaways

- Applications deploy as *modules* with *components*.
- The Module Framework manages installation, dependencies, and activation.
- Modules go through a *lifecycle* for simple management.

All applications are developed as modules, containing components, that live inside the Module Framework. To make it easier for the framework to easily handle installation, dependency resolution, and starting and stopping applications, there is a set *lifecycle* that all modules adhere to. *(Figure 2.9, page 50)*

Figure 2.9:



2.5.2 Module Installation

Takeaways

- On *install*:
  - The *Manifest* is validated.
  - Dependencies are resolved.
  - Module is set to *resolved*.

When a module is deployed, it enters the *installed* state. Here, the Module Framework checks the manifest for errors and resolves any external dependencies in the module. Once dependencies are resolved, the module moves into the *resolved* state.

2.5.3 Dependency Resolution

Takeaways

- On *resolved*:
  - Dependencies have been resolved.
  - Outside Java classes are available.
  - Any needed services are available.

- The Module can be *started*.

Once the module has been *resolved*, that means that any external Java classes, services, or other declared dependencies have been found and made available to the module. The module is now completely ready and can be *started*. Some modules may start automatically, and others may be started manually.

2.5.4 Active Modules

Takeaways

- Once *started*:
  - The Module becomes *active*.
  - Services can be called.
  - Applications can run.
- This is the main state for applications to execute.

After starting, the module is said to be *active*. Once a module is *active*, any services it provides can be used, any application components can be executed, and it can be relied upon as a dependency. This is the main phase of the lifecycle for your modules. Once they are active, nothing else happens in the module lifecycle until you stop the application.

2.5.5 Stopping Modules

Takeaways

- Running modules:
  - Can be *stopped*
  - Can be *uninstalled*

Once modules are *active* and running, you can turn them off by either *stopping* them or *uninstalling* them. Stopping a module simply halts execution and removes the module from active use. It can be easily restarted again. Uninstalling the module completely removes it from the Module Framework. You'll have to install it again to use it.

2.5.6 Lifecycle in Action

Takeaways

- See a module:
  - Deployed
  - Installed
  - Started
  - Active
  - Stopped
  - Uninstalled

Let's go ahead and do some hands-on work with a real module. We'll install a simple module, watch it being deployed and installed, and then start and stop it ourselves. We'll then see what it's like to uninstall a module once deployed.

Hands-on Exercise

- If you don't already have a Liferay bundle running:
  1. **Navigate** to `liferay-portal-7.0-[version]/tomcat-[version]/bin`.
  2. **Start** `startup.bat` to run Liferay.
     - On Linux or Mac OS X, open a terminal and run `catalina.sh run` from the same directory.
- Deploy the sample module:
  1. **Open** the exercise folder `exercises/02-application-dev/01-module-lifecycle`.
  2. **Copy** the module `com.liferay.training.lifecycle.command.jar`.
  3. **Paste** the JAR file into the `liferay-portal-7.0-[version]/deploy` directory.
- You should see output in your Liferay log window:

```
19:25:39,791 INFO [com.liferay.portal.kernel.deploy.
auto.AutoDeployScanner]
[ModuleAutoDeployListener:70] Module for /Users/
jonathonomahen/liferay-
portal-7.0-ce-b4/deploy/com.liferay.training.
lifecycle.command.jar copied
successfully. Deployment will start in a few seconds.
```

- After a few seconds, the module should start:

```
[Lifecycle Demo Module] This module is starting...
```

- The module is now active.
- The demo provides a shell command to test:

1. **Open** a terminal or *Command Prompt* window.
2. **Open** a telnet connection to the shell:

   ```
   telnet localhost 11311
   ```

3. You should be greeted with a prompt:

   ```
   g!
   ```

4. Test the new command by **typing** in `life` and **pressing** *return* on your keyboard.
5. You should see some promising output:

   ```
   [Lifecycle Module] Liferay: Enterprise. Open Source. For Life.
   [Lifecycle Module] This module is active and running!
   ```

- Test the stopping phase of the lifecycle:

1. **Enter** the command `lb` to find the module ID:

   ```
   434|Resolved    |     1|Lifecycle Demo (1.0.0.201601281922)
   ```

2. **Stop** the module with the command `stop [module id]`:

   ```
   stop 434
   ```

3. You will see a log message:

   ```
   [Lifecycle Demo Module] This module is stopping...
   ```

- You can easily restart the module manually as well:

1. **Start** the module with the command `start [module id]`:

   ```
   start 434
   ```

2. You will see a shell message:

   ```
   [Lifecycle Demo Module] This module is starting...
   ```

- The module will also stop when you uninstall it:

1. **Navigate** to `liferay-portal-7.0-[version]/osgi/modules`.
2. **Delete** the JAR file `com.liferay.training.lifecycle.command.jar`.
3. You will see the log message:

   ```
   [Lifecycle Demo Module] This module is stopping...
   ```

2.5.7 In The Real World

Takeaways

- The Module lifecycle assists debugging.
- Easy to implement
- Used in all modules

The module lifecycle helps let you know what's going on in the Module Framework. This way, you have an idea of whether or not your modules are installed, running, or stopped. It'll also help you determine if you've got any problems starting, which helps you with debugging runtime issues. Every module implements the lifecycle, so you can be confident that every module starts in the same way.

2.6 Building a Module

2.6.1 Module Framework in Practice

Takeaways

- Building a module is easy:
  - **Create** a Java project.
  - **Create** any necessary Java objects.

  - **Create** a *manifest*.
  - **Export** the project (with compiled classes) in a JAR.
- Once built, **deploy** in `deploy` folder.

Nothing gets you acquainted with the Module Framework quite like using it. We're going to look at what it takes to create a *module* that can be deployed in Liferay from scratch. Given what we know about what makes a module tick, there's very little we need to provide to make a basic module. To have a valid module, we need a basic Java project that contains any Java objects we need, a *manifest* file, and a simple folder structure. Once we've built all of that, we can create the JAR that encapsulates all of our files, and deploy it by placing it in Liferay's `deploy` folder.

2.6.2 Artisan Module Development

Takeaways

- No special tools to build
- Simple Java EE dev environment
- Write a few files
- Build, JAR, and deploy

Writing a module by hand is not nearly as intimidating as it sounds. Unlike many other applications like servlets or portlets, there are relatively few pieces that come together to make a module. There are no special tools necessary to complete a basic module project, aside from a Java development environment, and a running Liferay instance to test deployment in. We'll set up a basic development environment, create a project, write a few files, build, JAR, and deploy into Liferay.

2.6.3 Setting Up the Dev Environment

Takeaways

- **Install** the Eclipse Mars JEE edition IDE:

  ```
  https://eclipse.org/downloads/packages/
  eclipse-ide-java-ee-developers/marsm1
  ```

- **Install** a Liferay 7 bundle:

  `https://dev.liferay.com/web/liferay-7-community-expedition/`

- **Startup** Liferay and complete setup.

A very basic development environment just requires a standard Java EE environment and Liferay. If you haven't already, go to the Eclipse project Site to download the Mars Java EE edition of the IDE. You can use any IDE for Java development when building projects for Liferay, but the exercises that follow will assume you have the same setup here. Since we already have a Liferay instance setup, the only additional installation you need is Eclipse Mars JEE edition.

2.6.4 Creating the Project

Hands-On Exercise

- Create the basic Java project:

  1. In Eclipse, **click** on the menu *File* → *New* → *Project...* → *Java Project*.
  2. In *Project name*, **enter** *com.liferay.training.service.hello*.
  3. **Click** *Next*.

- Set the build path to include the basic Module Framework API:

  1. **Click** on the *Libraries* tab.
  2. In the window below, **click** the button *Add External JARs...*.
  3. **Choose** the JAR file:

     `liferay-portal-7.0-[version]/osgi/core/org.eclipse.osgi.jar`

  4. **Click** *Finish*.

2.6.5 Implementing the Lifecycle

Takeaways

- All modules implement the basic lifecycle.
- A module needs to:

  - Install
  - Resolve
  - Start
  - Stop
  - Uninstall

- The Module Framework does *install*, *resolve*, *uninstall*.
- Each module handles *start* and *stop*.
  - Handled by an activator object
  - Implements methods from `BundleActivator`

For our module to deploy and live within the framework, it needs to go through the basic lifecycle: *install*, *resolve*, *start*, *stop*, and *uninstall*. The Module Framework provides the essential implementation for *install*, *resolve*, and *uninstall*. Since each application can vary in what kind of setup and configuration it wants to do, every module implements its own *start* and *stop*. This is similar to lifecycle phases like *init* and *destroy* in servlets and portlets.

Hands-On Exercise

- Create the module's activator class:

1. **Right-click** on the project in your *Package Explorer*.
2. **Click** *New* → *Class...* on the menu.
3. **Enter** `com.liferay.training.service.hello.module` for the *Package*.
4. **Enter** `HelloActivator` for the *Class*.
5. Under *Interfaces*, **add** the `BundleActivator` interface.
6. **Click** *Finish*.
7. **Insert** the snippet *01 HelloActivator methods* into the body of the class.
8. **Save** the file.

Announcing the Module

Takeaways

- Every module needs a *manifest*.
- Manifest (`MANIFEST.MF`) describes:
  - **symbolic name:** the module's name (`Bundle-SymbolicName`)
  - **version:** the module's version (`Bundle-Version`)
  - **name** the module's human–readable name (`Bundle-Name`)
- There are more options available, like dependencies, service packages, etc.

Every module needs to be packed up with a *manifest* file. Just like the bill of goods in a shipment, the manifest describes what is in the module, and that is in a proper module. This is where we set the *symbolic name*, the *version*, and more. We'll also declare our activator class: `HelloActivator`.

Hands-On Exercise

- First, we'll create the proper folder structure:
  1. **Right-click** on the top of your project in the *Package Explorer*.
  2. **Click** on *New → Folder...* in the menu.
  3. **Create** a new folder named `META-INF` at the root directory of your project.
- Create the manifest file:
  1. **Create** a new file named `MANIFEST.MF` in the `META-INF` folder.
  2. **Insert** the snippet *02 Manifest File* in the empty file.
  3. **Save** the manifest file.

Packing it Up

Hands-On Exercise

- Package the finished product:
  1. **Right-click** on the project.

2. **Click** on *Export...* in the menu.
3. **Select** *Java* → *JAR file* as the export type.
4. **Click** *Next*.
5. **Browse** for the location of the *JAR File* export.
6. **Name** the JAR file `com.liferay.training.service.hello.jar`.
7. **Click** *Next*, and leave the defaults.
8. **Click** *Next*.
9. Under *Specify the manifest*, **choose** *Use existing manifest from workspace*.
10. **Click** *Browse...* next to *Manifest:*, and locate the `MANIFEST.MF` file in your project.
11. **Click** *Finish* to export the JAR.

Is it Active?

- After you've installed your JAR file, you can check on its status:

1. **Open** a terminal and **telnet** on `localhost:11311`.
2. **Use** the command `lb` to find your installed module:

```
423| STARTING| com.liferay.training.service.hello.jar:1.0.0
```

- Is it active?
- Our `BundleActivator` has a default activation policy of *lazy*.
- *Lazy* activation means that it won't be started unless it's needed.
- We can also start it manually with the `start` command:

```
g!> start 423
...
g!> lb
423| ACTIVE| com.liferay.training.service.hello.jar:1.0.0
```

2.7 Building Services

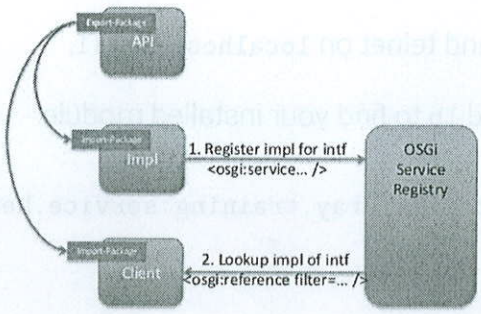
2.7.1 Service Architecture

- Services in the Module Framework are implemented as a series of Java classes and registered with the *Service Registry*.
- Modules that contain services register the services through the 'BundleContext' in the module's 'BundleActivator'.
- The service registry decouples the implementation of the service from its API, allowing multiple implementations to be deployed.
- Modules can request specific services by name and version in their *manifest*:

```
Import-Package: com.liferay.portal.service.UserService;version=1.2.1+
```

(Figure 2.10, page 60)

Figure 2.10:



2.7.2 Creating a Basic Service

- Creating a service requires:
  - Building an interface
  - Creating an implementation
- You can create a service API that is published to the rest of the framework and keep the implementation private to the module.

2.7.3 Implementing Your Service

- When implementing the service, you need to:
  - Publish the service to the module framework
  - Export the service in the module's *manifest*
- Publishing the service is done during the *start* phase of the module lifecycle:
  - In the module's `BundleActivator`, register the service with `registerService(Service.class.getName() , serviceImpl, null)`
  - When the module is started, the service implementation will become available via the service registry .

2.7.4 Hands-On Exericse

Creating A BND Project

1. As we've done before, let's first build the module.
2. In IDE, **click** File → New → Bndtools Project.
3. **Select** *Empty* under the Bndtools folder.

Naming The Project

1. **Click** Next.
2. **Name** the project `hello-api`.
3. **Click** Finish.

Creating The Interface

1. **Right-click** the `src` folder.
2. **Select** New → Interface.
3. **Name** the interface `HelloService`.

Adding The Methods

1. Within the interface, **add** the snippet `01-HelloService API`.

```
public String say();

public String say(String response);
```

2. **Save** the file.

Enter the Module Version

1. **Open** `bnd.bnd`.
2. In the version field, **enter** `1.0.0.${tstamp}`.
3. **Click** on the *Source* tab.

The Manifest

- In the source of `bnd.bnd`, you'll notice the manifest header `Bundle-Version: 1.0.0.${tstamp}`.
  - `${tstamp}` will add a timestamp based on the date the module is created.
- Rather than creating the manifest all by hand, we can have Bndtools create the manifest for us.
- Bndtools will take whatever is in the content and description tab and apply the correct manifest headings to create the manifest file for us.
- Unless we really need to, there is no longer a need to modify the source of the manifest directly.

Exposing Our Interface

1. Next to the heading `Export Packages`, **click** on the green plus button.
2. **Add** the `com.liferay.training.service.hello` package.
   - If you receive a Missing Package Info window, **click** OK.
3. **Save** the `bnd.bnd` file.

The Source, One Last Time

- If you examine the source after adding the export package you'll see:

```
Export-Package: com.liferay.training.service.hello
```

- These will be added to what will become `MANIFEST.MF`.
- Bndtools also keeps an updated JAR file based on all the work we are doing.

Sending it Off

- Let's now deploy the API Module.
- Bndtools has the generated JAR file found in the `generated` folder.
- Let's go ahead and get the JAR file and deploy it.

1. **Expand** the `generated` folder.
2. **Right-click** the `hello-api.jar`.
3. **Select** Show In → System Explorer.
    - Alternatively, you can navigate to the project folder located in your workspace to find the `generated` folder.

Into the Module Framework

- Your file explorer should have opened a folder that contains the API Module.

1. **Copy** the API Module.
2. **Paste** the API Module into the `deploy` folder.

Checking Up On The API Module

- Our API Module should be in the Module Framework, but let's check to make sure.

1. **Open** a terminal/Command Line and **telnet** into the shell using `telnet localhost 11311`.
2. **Type** `lb` to see a list of modules installed. At the bottom will be our API Module.

What Did We Just Do?

- We created a project using Bndtools, a development environment that aids in creating modules.
- Next, we created the API for our services.
- Lastly, we modified the `bnd.bnd` file, which creates the manifest for us based on the inputs we enter in `bnd.bnd`.

The Implementation Module

- The Implementation Module will be implementing the API module that we created previously.
- We'll then import the package from the API Module.
- Typically, next the Implementation Module will publish the Implementation of the service into the Service Registry.

The Service Registry

- The Service Registry is the 411 of the Module Framework.
- A module can "call" the Service Registry for a specific implementation of a service.
- A module can also let the Service Registry know of an implementation so that it can be called upon.

Implementing The Implementation Of The Implementation Module

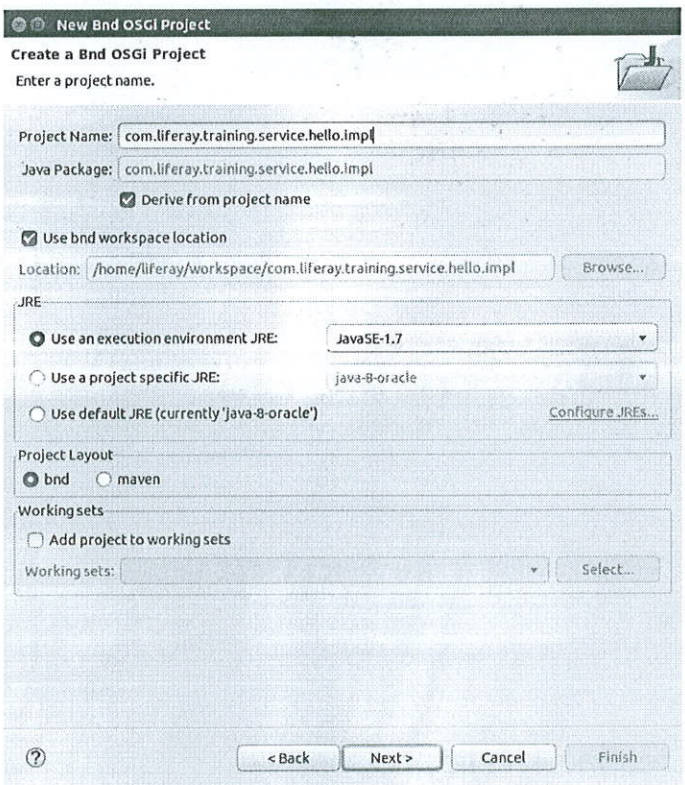
- We'll first create the implementation class for the API module.
- Next, we'll register our service to the Service Registry.
- Finally, the Implementation Module will import the package from the API module.
- That's it! Let's do... or do not... there is no try.

Exercise: Creating The Implementation Bundle

1. **Click** File → New → BND Tools Project.
2. **Select** Empty followed by Next.
3. **Name** the project `hello-service`.
4. **Click** Next.

(Figure 2.11, page 65)

Figure 2.11:



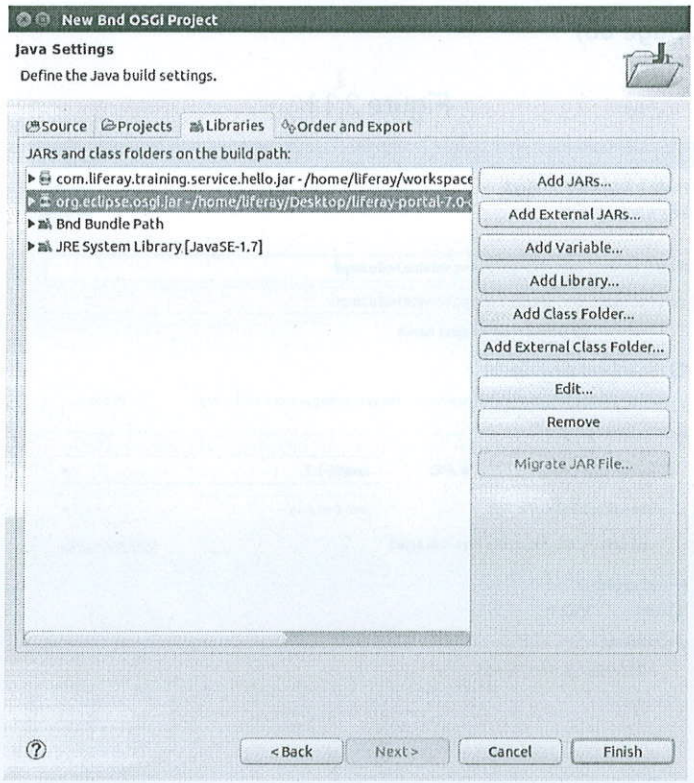
Exercise: Importing Libraries

1. **Click** on the *Libraries* tab.
2. **Click** on *Add Exernal JARs...*

3. Add the `org.eclipse.osgi.jar` found in `/LIFERAY-HOME/osgi/core`.
4. Add the `hello-api.jar` found in `workspace/hello-api/generated/`.
5. Click Finish.

(Figure 2.12, page 66)

Figure 2.12:



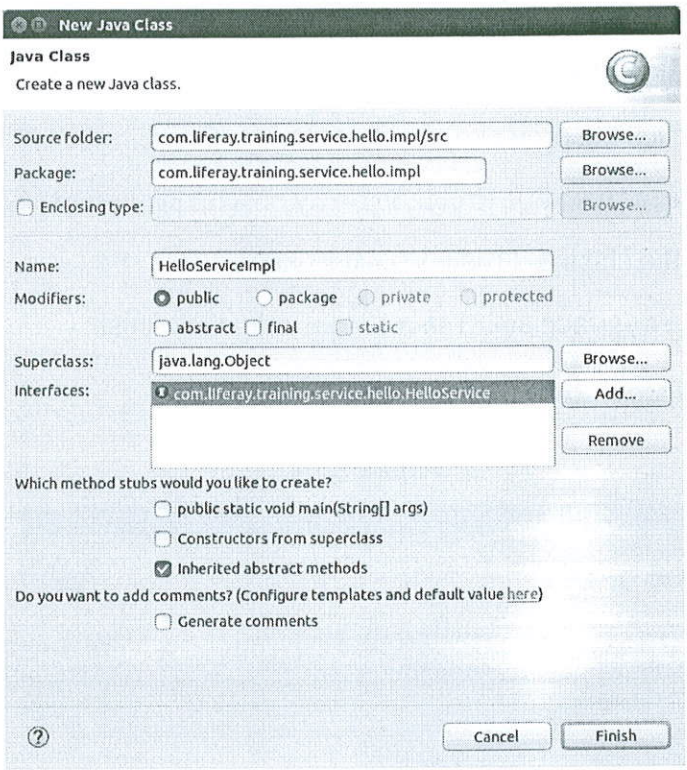
Exercise: Creating The Impl Class

1. Right-click the project `hello-service`, New → Class.
2. Name the class `HelloServiceImpl`.
3. Set the package to *com.liferay.training.service.hello.impl*
4. Add the `HelloService` as an interface.

5. **Click** Finish.

(Figure 2.13, page 67)

Figure 2.13:



Exercise: Must Have Called A Thousand Times

1. Inside of the `HelloServiceImpl` class, **overwrite** the auto-generated methods with snippet *01-HelloService Implementation*.
2. **Save** the file again.
3. If we call `say()` we get one message and if we pass *hello* we'll get another message. It's very simple.

Registering The Service

- To register the service, this will be done when the module is started up.

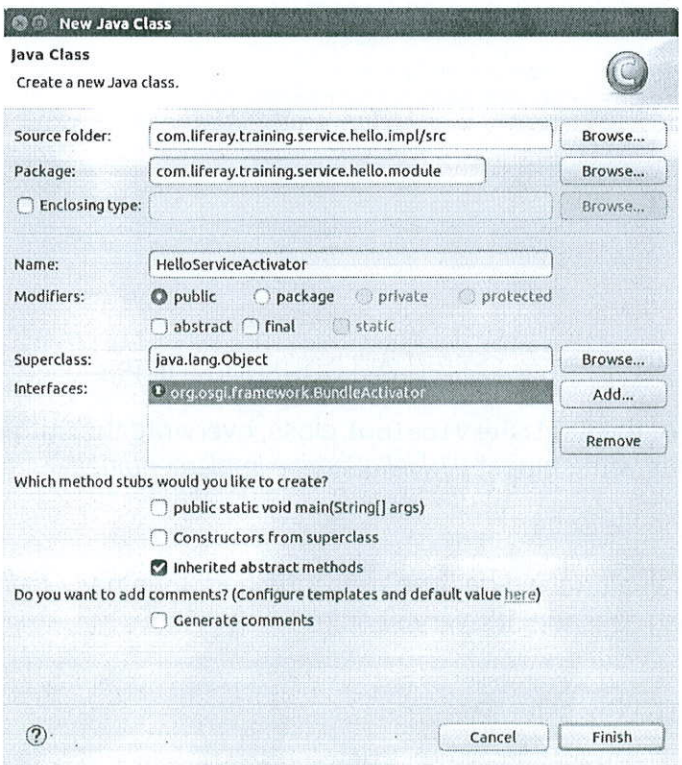
- The `BundleActivator` object is what takes care of the start up of module.
  - Was it mentioned that bundle is the OSGi term for module?
- Onwards to creating the BundleActivator we go.

Exercise: Registering The Service

1. **Right-click** the project and select New → Class.
2. **Change** the package to `com.liferay.training .service.hello.module`.
3. **Name** the class `HelloServiceActivator`.
4. **Add** the Interface `BundleActivator` and **click** finish.

(Figure 2.14, page 68)

Figure 2.14:



Exercise: Do You Want To Start A Module?

1. **Override** everything within the class with snippet *02-Register-Service-Method*.

2. **Resolve** imports using:

    - `CRTL/COMMAND + SHIFT + O`

3. **Save** the file.

- The `registerService()` allows us to register a service object (`_helloService`), by its name (`HelloService.class.getName()`).

- We can also pass properties of the service as well.

Importing The Api Module

- The last and final step is to import the API Module.

- This is done within the bnd.bnd file.

- The `bnd.bnd` file is the file that BndTools uses to make the `MANIFEST.MF` for us.

Exercise: Keep Things Hidden (Just In Case)

1. Within the `bnd.bnd`, under Version: **add** *1.0.0.${tstamp}*.

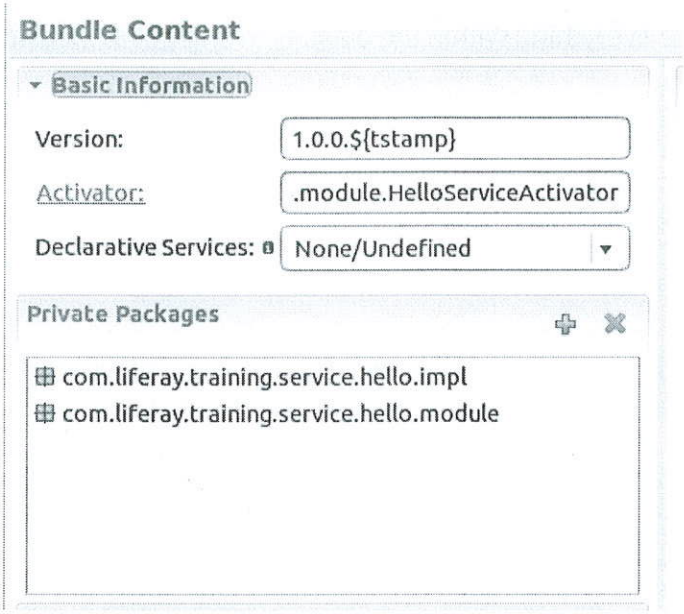
2. For the Activator **add** `com.liferay.training.service.hello.module.HelloServiceActivator`.

3. Under *Private Packages* **ensure** that our two packages, `com.liferay.training.service.hello.impl`, and `com.liferay.training.service.module` are included.

    - If they are not included, **click** on the *green plus* next to *Private Packages* to add them.

(Figure 2.15, page 70)

Figure 2.15:



Exercise: No Tariffs Here

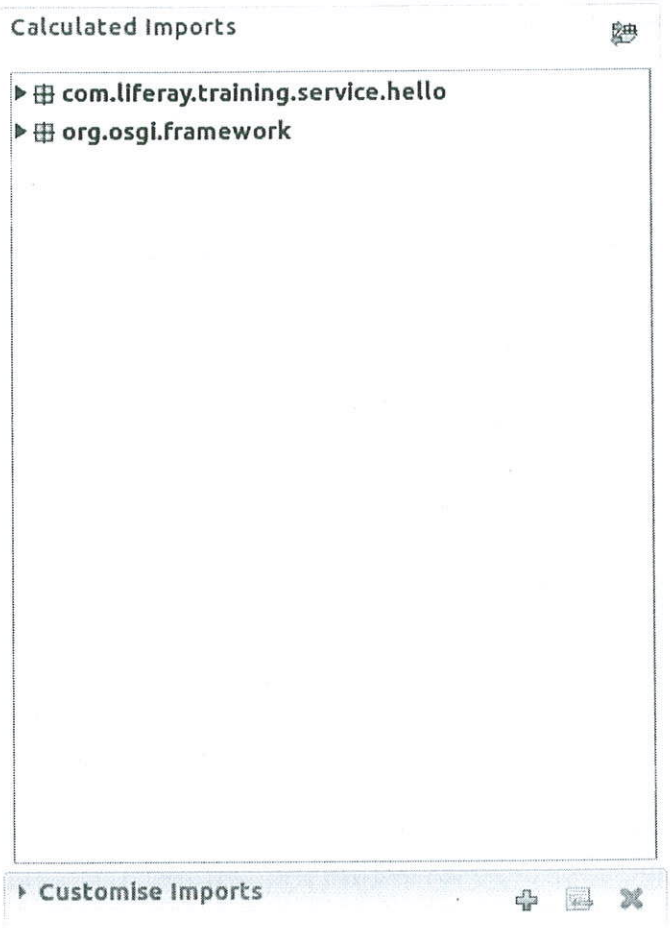
1. **Confirm** `com.liferay.training.service.hello` and `org.osgi.framework` are in *Calculated Imports*.
2. Under *Customise Imports* **Click** on the *green plus*.
3. In the *Pattern* field **add** `com.liferay.training.service.hello`.
4. **Save** the file.

(Figure 2.16, page 71)

Exercise: Off You Go, Into The Module Framework

1. Now that we have finished creating our Implementation Module, let's send the JAR file off to summer camp (or the Module Framework).
2. **Copy** the JAR file from the `generated` folder.
    - Found in `workspace/hello-service/generated`
3. **Paste** the JAR file in the `/deploy/` folder.

Figure 2.16:



Exercise: I Must Have Called A Thousand Times

1. To test the service, **deploy** the `com.liferay.training.service.hello.test`.
2. **Open** a terminal/Command Prompt and telnet into the shell.
    - `telnet localhost 11311`
3. **Confirm** the API and Implementation Module has been deployed using *lb*.
4. **Type** *say* or *say hello* to test the service call.

```
g! say
```

```
Hello...


g! say hello
It's me...


g! say hello
I was wondering if after all these years you'd like to meet.
```

To Go Over Everything

- The Implementation Module is the second module that makes up the OSGi Service.
- Like any service architecture, it implements an API, in this case the API Module.
- To register a service in the Service Registry, we use method call `registerService()` on the BundleContext Object within the BundleActivator.
- The Implementation Module has to import the API Module.

2.8 The JSR-286 Portlet Specification

2.8.1 Overview

- In the Module Framework, applications are composed of modules that can interact with each other.
- Modules contain components, each of which has different responsibilities and functions.
- For example, the MVC design pattern can be implemented using a separate module for each of the three different layers.
- The View layer can be implemented using one module, the Controller using another module, and the Model using a third module.

2.8.2 Portlets and Components

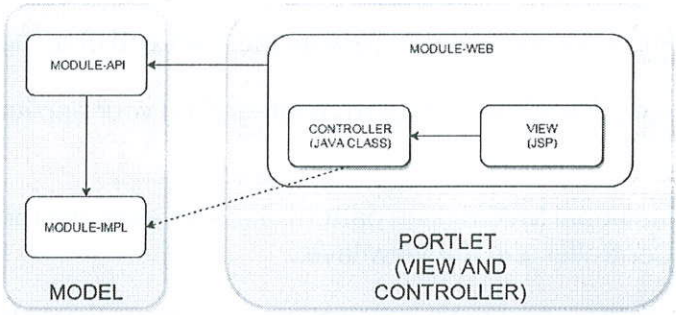
- A portlet is a component that acts as the front-end of an application.
- Portlets are the primary method of integrating your application with Liferay's UI.
- In the traditional MVC design pattern, a portlet adds the functionality for the Controller and the View layer.
- The Controller provided by a portlet can range in complexity from something very basic (switching between pages of a portlet) to more advanced functionality (handling actions or processing data).
- Portlets can also take the middle ground of having a main Controller that hands off to a separate class for the data processing/business logic.

2.8.3 Portlets In a Modular World

- In addition to being a Module Framework, Liferay also includes a portlet container.
- The presence of this portlet container allows applications to integrate with Liferay's UI and to be displayed on its web interface.
- Therefore, any application you want to integrate with Liferay's UI needs to have a portlet component.
- Modular MVC-based applications are typically made up of three modules:
- `<module-name>-api` - This module contains the service API.
- `<module-name>-impl` - This module contains the service implementation.
- `<module-name>-web` - This module contains the portlet component, with the View and Controller layers of your application.

(Figure 2.17, page 74)

Figure 2.17:



2.8.4 Do I Need a Portlet?

- You need a portlet if you want your application to have a UI and show up as an application in Liferay's web interface.
- Since most applications do require a UI, it's very likely that your application will require a portlet component as well.

2.8.5 Example of a Portlet

- Let's take a look at an application that could use a portlet.
- For our example, let's consider an MVC application that needs to keep track of a collection of books.
- The model would be implemented as a service that would connect to the database to store/retrieve the information about the books in your collection.
- Following the convention we've looked at, the model would be implemented as two modules, `book-api` and `book-impl`.
- We also want to be able to view our collection, which is where a portlet comes into play.
- This would be implemented as a module called `book-web`, and would contain the View and Controller of our application.

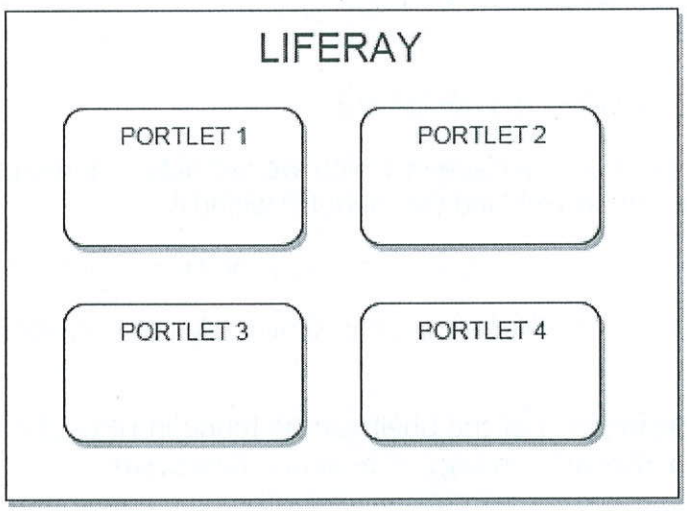
2.8.6 How Do Portlets Work?

- Portlets are meant to coexist with other portlets on the same page.

- This means that portlets can interact with each other, like any other components. It also means that a portlet may be run due to an action a user has performed in another portlet.
- Just as modules have a series of states they go through (called a lifecycle), portlets have their own lifecycles as well.
- This helps portlets better manage different needs for the application, such as displaying content, processing data, etc.

(Figure 2.18, page 75)

Figure 2.18:



2.8.7 The Java Standard

- Portlets are governed using a Java standard called JSR-286 (Portlet 2.0).
- Developing your portlet using this standard allows you to easily move your portlet to any other portlet container that complies.
- Portlets developed for the Module Framework, while not strictly standards-compliant, use the same concepts as are provided by the standard and are structured very similarly.
- Because of its importance, we will be taking a look at how the Java standard works.

- If you're interested, more information on the standard can be found at `https://jcp.org/en/jsr/detail?id=286`

2.9 Interacting with the Shell

2.9.1 Introduction to the Shell

- Now that we have an idea of what's inside the Module Framework and how it works, we need a way to be able to launch, configure, and control the modules within the Module Framework.
- Let's take a look.

2.9.2 Feature of the Shell

- The shell is an interface in which we are able to interact with the Module Framework and the modules within it.
- The shell is lightweight, and it's easy to add new commands.
- It acts like a POSIX shell and has similar BASH/SH syntax and commands.
- The specification of the Shell can be found in RFC-132: `https://osgi.org/download/osgi-4.2-early-draft.pdf`

2.9.3 Connecting to the Shell

- The Shell is composed of four parts:
  1. The Command Processor
  2. The ThreadIO
  3. The Converter
  4. The Command Provider
- The Command Processor is what allows outside access to the shell through telnet or other means.
- We first encountered the Command Processor when telneting into the shell using the command `telnet localhost 11311`.

2.9.4 Entering a Command

- Once connected to the shell, the Command Processor is also responsible for the commands that we enter, making sure the right command is executed.
- Everything in the shell is Object-based manipulation rather than String-based.
  - The commands we execute are method calls to particular services which implement the command.
- When we type a command such as `start 431`, it's actually a method call on a specific service that corresponds to the command start.
- The command services and their implementations are found within the Command Provider.

2.9.5 User Interaction

- If a specific command needs to interact with the User, that's where the `ThreadIO` comes into play.
- The `ThreadIO` provides the ability to use `System.In`, `System.Out` and `System.Err` to prompt for an input or send messages to the end User.
- If `ThreadIO` prompts the User for an input or there is already a parameter following the command (e.g. `431` in the command `start 431`), the parameter is taken and passed to the last part of the shell.

2.9.6 From Text to Objects

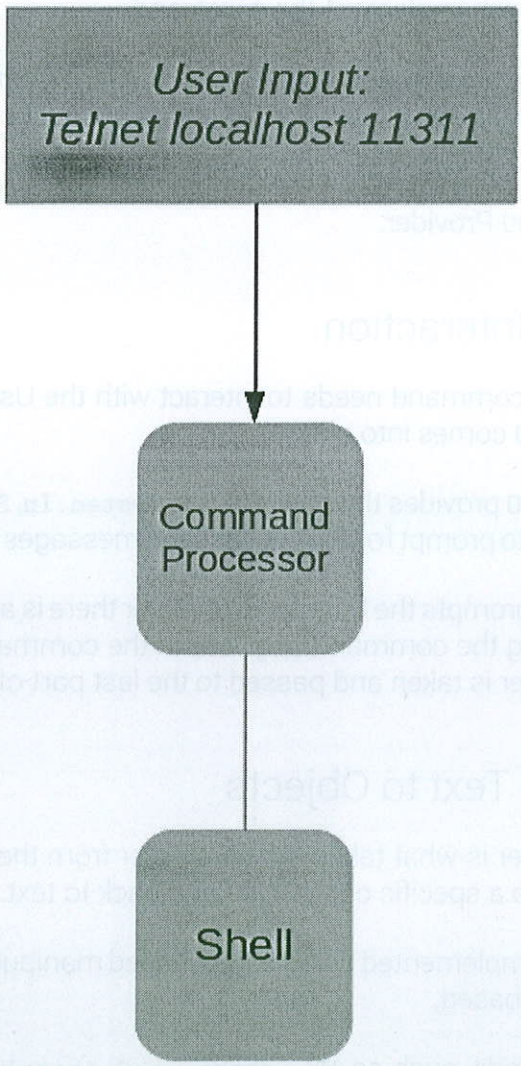
- The Converter is what takes the parameter from the command and converts it to a specific object-type and back to text.
- The shell is implemented using object-based manipulation rather than String (text) based.
- The commands, such as `lb, start, and install` correspond to methods of a service.

2.9.7 Telnet Through the Command Processor

- Let's take a look at everything all put together.
- We first telnet into the shell through the Command Processor.

(Figure 2.19, page 78)

Figure 2.19:



- When we type a command such as `start 431`, the Command Processor takes the command and invokes the method that corresponds to that command found in the Command Provider.

(Figure 2.20, page 95)

- At this point if the command needs an input from the User or needs to display a message, the ThreadIO provides that for the command.

(Figure 2.21, page 96)

- Since our command `start 431` has the parameter of 431, the Converter will typecast our parameter into the specific object type it needs to be in order to run our command (method).

(Figure 2.22, page 97)

- Finally, the command is executed, and, if applicable, feedback is sent back to the end User.

(Figure 2.23, page 98)

2.9.8 Tiny Shell Language

- The Tiny Shell Language (TSL) is the command syntax of the shell interface that makes all this possible.
- TSL is what allows text we input to be interpreted as method calls on Java `Objects`.
- TSL allows for piping, closures, variable setting and referencing, and other features.
- For now, we'll keep things simple.

2.9.9 Commands of the Shell

- Starting off, to get a list of commands simply type `help`.
- With each command, you can type `man <command>` to get the manual or help guide of a specific command (assuming that one is written for it).

- With these two commands, you can read about each command and figure out what all of them do, but we'll spare you the reading and show some of the common commands.

2.9.10 List of Modules and Piping

- To show a list of all the *modules* installed within the Module Framework, type `lb`.
  - `lb` stands for list *bundles*, Liferay modules are known as bundles in OSGi.
- If you are looking for a specific module, you can use `lb <bundle name>`.
- Commands can be chained or piped using | allowing one command's input to be the input of another command.

```
_g! lb | grep Polls_
305|Active      |    1|Liferay Polls Service (1.0.0)
329|Active      |    1|Liferay Polls Web (1.0.0)
398|Active      |    1|Liferay Polls API (1.0.0)
true
```

2.9.11 Variable Assignment

- We can assign values to a variable just like we do in Java.

```
variable1 = ' Here is a Sentence'
```

- We can also reference our variables using `$<variable-name>`.

```
g! echo $variable1
Here is a Sentence
```

2.10 Building Portlet Modules

- Portlet modules are the same as other modules.
- The only difference in creating a portlet is a special type of component.

2.10.1 Portlet Components

- Components are created by declaring that some of your Java objects are *components*.
- A component is a special type of object that can execute specific functionality:
  - Services
  - Portlets
  - Servlets
- Often, components don't need to be explicitly activated by a `BundleActivator`, meaning we have one less Java object to create.
- To create a component, we can configure it by hand in an XML file, or we can use *Declarative Services*.
- *Declarative Services*, DS for short, provide a number of annotations to make it easier to create new modules and components with less code.
- To use DS, we can shortcut writing a long manifest file and configurations by using a build tool called *Bndtools*.
- *Bndtools* provides a new file, `bnd.bnd` to set a few settings in.
- Once configured, *Bndtools* will generate the manifest, package the module, and have it ready to deploy.
- To create a service with DS using Bndtools, we only need to specify we are using DS in our `bnd.bnd`:

```
-dsannotations: *
```

- This allows us to leverage annotations so that all we have to do to implement a service is:

```
@Component (
service = MyService.class
)
public class MyService {
...
}
```

- With only one class file, and a `bnd.bnd`, Bndtools creates everything else needed to deploy the module.
- Portlets are created easily by specifying a new component.
- Portlet components are different in that:
  - They implement the `Portlet` interface
  - The contain one or more portlet lifecycle methods: `render()`, `doView()`, `processAction()`, etc.
- Declaring a portlet component only requires:

```
@Component (
service = Portlet.class,
property = {
    "javax.portlet.display-name=My Portlet"
}
public class MyPortlet implements Portlet {
...
}
```

- Portlet components are deployed in the same way as other modules and are no different from other components.

2.10.2 Configuring Portlet Attributes

- Basic portlet configuration is done through *properties*, which are set through the annotation:

```
property = {
  ...
}
```

- All properties that are available correspond to what can be set in the `portlet.xml`, `liferay-portlet.xml`, or `liferay-display.xml`.
- Properties coming from the portlet standard are prefixed with `javax.portlet`.
- Properties specific to Liferay are prefixed with `com.liferay.portlet`.

2.10.3 Presentation Layer

- Implementing the View layer in a portlet component can be as simple as including some JSPs.
- JSPs and other non-code resources should be contained in their own folder structure, such as `resources/html`.
- The module should declare these resources, so that the framework can make them available to the running code.
  - In `bnd.bnd` you can set `-useresource: resource/` to include directories or specific files as resources.
  - Once the JSPs are in their proper location, you can configure your portlet to use them.
- We can use Liferay MVC's portlet class `MVCPortlet` to shortcut basic JSP configuration.
- MVCPortlet will autowire display of a JSP configured through the portlet's `init-params`:

```
@Component (
service = Portlet.class,
property = {
    "javax.portlet.display-name=My Portlet",
    "javax.portlet.init-param.view-template=/html/view.jsp"
}
public class MyPortlet implements MVCPortlet {

}
```

2.10.4 Controller Layer

- Implementing the Controller layer is as simple as adding new methods in your portlet component's class:

```
@Component (
service = Portlet.class,
property = {
    "javax.portlet.display-name=My Portlet"
}
```

```
public class MyPortlet implements Portlet {

    @ProcessAction(name="myaction")
    public void myaction() {
    ...
    }
}
```

2.10.5 Discovering Services

- Consuming services in the module framework is a matter of asking the container for the appropriate service
- The basic way is to create a `ServiceTracker` object to listen to the framework's service registry
  - When the service is installed or active, the registry will use the service tracker's setter methods to hand over an instance of the service
- This requires additional code, since you would need to track every external service you wantto use
- Instead, using *Declarative Services*, we only need to provide a setter method for the service wherever we want to use it:

```
@Reference
protected void setMyService(MyService serviceImpl) {
...
}
```

- For each service we want to use, we only need to add a new `@Reference` annotation for it. You can add as many services as you want.

2.10.6 Integrating with Liferay Frameworks

- Many Liferay framework integrations are a matter of:
  - Creating a class that extends a base class
  - Registering that class through a portlet property

- Since we can easily set component properties in the `@Component` annotation, much of the work in integrating with the various frameworks is a matter of implementing proper Java classes.

2.10.7 Search

- A common application need is to have some or all of your content searchable
- Liferay's Search API provides a simple way to integrate with the platform–wide search capability
- Any application can add data to the search index by implementing a Search Indexer
- To create an indexer, create a new Java class that extends

  `com.liferay.portal.kernel.search.BaseIndexer`

- In the indexer, you can add data to the indexer `Document`
- To publish the indexer to the platform, you need to set the property on your portlet component:

```
property = {
  "com.liferay.portlet.indexer-class=com.test.indexer.MyIndexer"
  }
```

- To use the indexer, you can call the `IndexerRegistryUtil` to retrieve the indexer, and call the `reindex()` method to update the index

2.11 Debugging Module Deployment

- Errors that occur on deployment can most commonly be diagnosed using the shell.
- From within the shell, using the `lb` command will give you the status of any module.
- You can also get a more compact list using the `ss` command.
- If a bundle is listed as only `Installed`, then it probably has a resolution error.

- You can discover what a module is missing by using the `diag` command.

2.11.1 Dependency Resolution

- Modules declare their dependencies on:
  - Specific Java packages and services
  - Specific bundle (module) names
- When the framework installs a module, it attempts to resolve dependencies.
- Dependencies cannot be satisfied by relying on class loader hierarchy.
- In the module framework, there is almost no class loader structure: each module has its own class loader.
  - Module class loaders by default only have access to the standard Java APIs in `java.*` or the standard module framework container API in `org.osgi.*`.
- When the dependency is found, the module's class loader is given the reference to the class name and bundle it can be found in.
  - At runtime, class instantiation is very fast — the class loader doesn't have to look for the class; it's already mapped in a simple registry.
- If the dependency resolver *cannot* locate a required package or bundle (module), it will *not* start or activate the module.

2.11.2 Troubleshooting Deployment

- Most deployment issues are a syntax or configuration error in the manifest, or unresolved dependencies.
- In both cases, using the `diag` command will give you information to resolve the error.
- When redeploying a patched module, if the version number has not changed or the loaded module becomes stale, you can force it to reload from the JAR file by using the `refresh` command.

- This will also start/stop the module.

2.12 Upgrading WARs

2.12.1 Overview

- As we've learned by this point, Liferay applications are composed of one or more modules.
- This module structure is a step away from the traditional application structure used in previous versions of Liferay.
- The new version of Liferay, however, does retain backwards compatibility.
- Even though we've transitioned to a more modular architecture, Liferay still supports the traditional WAR-style applications that you may already be familiar with.

2.12.2 How do I upgrade my plugins?

- The upgrade process is divided into two big steps:
  - Converting your existing 6.2 plugin to a plugin for this new Liferay version:
    * At this point, we're still keeping it as a WAR-style application.
    * This stage should help iron out any API or breaking changes.
  - Converting your WAR application to modules:
    * This step is optional; we will see why it's highly recommended.

2.12.3 Upgrading my plugins to the new API

- Your first task is to upgrade your plugins to adapt them to the new Liferay API.
- With that purpose, a new tool called **Breaking API Migration Tool** has been included in the Liferay IDE 3.0 to help you in this task.

- As you know, Liferay IDE is a plugin for Eclipse, but you can use plugins SDK and *blade tools* (which includes a migration tool as a command line application) to upgrade your plugins if you prefer not to use Eclipse. To know more about how to upgrade your plugins manually, please check the section, *How can my plugins run on Liferay?* You can find more information about blade tools in the Liferay Developer Network:

  `https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/introduction-to-blade-tools`

2.12.4 Breaking API Migration Tool

- You can use this tool to upgrade your plugins from 6.2 to 7.0.
- It analyzes your code to find migration issues and suggests a solution for them, explaining the reason for the change. It can also autocorrect many of those issues.
- The tool provides the UI in the IDE for this workflow migration.
- This tool is based on the breaking changes document, found here: `https://github.com/liferay/liferay-portal/blob/master/readme/7.0/BREAKING_CHANGES.markdown`. It is auto-updating and extensible with new changes.

2.12.5 Using Breaking API Migration Tool

- After executing the migration tool on your plugins, a new UI will show you the upgrade issues and how to solve them:

(Figure 2.24, page 98)

2.12.6 Using Breaking API Migration Tool

- You will be able to check the list of issues:

(Figure 2.25, page 99) *(Figure 2.26, page 99)*

2.12.7 Using Breaking API Migration Tool

- If you click on a class or resource, you will be able to check the list of migration issues for that object. You can also mark as "resolved" the issues you fix.

(Figure 2.27, page 99)

2.12.8 Using Breaking API Migration Tool

- For every issue, the tool will display the following information:
  - **What changed?**: This explains the modifications in the new API.
  - **Who is affected?**: Here you can figure out what objects need to be reviewed.
  - **How should I update my code?**: This explains the changes you have to make in your code in order to update it properly. This also shows a few examples, depending on the case.
  - **Why was this change made?**: Learn the reasons for making the modification in the new API.
- In some cases, you can put the cursor over the error in the code to check possible solutions and select one of them to fix the code automatically.

2.12.9 Using Breaking API Migration Tool

(Figure 2.28, page 99)

2.12.10 Should I Go Modular?

- As previously mentioned, converting your application to a module is not required.
- So when should you convert?

2.12.11 When to Convert?

- If you have a large application with hundreds or thousands of lines of code and many developers working on it concurrently, splitting it into modules will provide agility by allowing for more frequent releases.
- You should convert if your application has parts that you want to consume from elsewhere. Because modules make dependency management easier than traditional WAR-style applications, this makes sharing just a portion of your application very easy and allows you to reuse existing code.

2.12.12 When Not to Convert?

- If you have a portlet that's JSR-286 compatible and you want to retain the ability to deploy it into another portlet container
- If you're using a framework heavily tied to the more traditional Java EE programming model

2.12.13 Why Should I Convert?

- Going modular allows you to split up your application into smaller sections that are much easier to manage.
- This model also allows for incremental release cycles, as modules can be updated independently of each other.
- For example, if a JSP needs to be changed due to a security issue, the client module can be updated without needing to touch the persistence module.
- Dependency management is simplified, as the module framework explicitly lists all of its dependencies and will refuse to run unless they are satisfied, eliminating obscure runtime errors that may otherwise occur.
- **Going modular also allows for better integration into the Liferay platform and makes it easier to communicate with other modules installed in the Module Framework.**

2.12.14 Can I Develop Using WARs?

- WARs, by definition, are not modular, since they don't dynamically manage their dependencies and imports/exports.
- In Liferay DXP, we're moving towards greater modularity with the new Module Framework.
- To achieve this, all plugins are now deployed as modules.

2.12.15 Liferay's Compatibility Layer

- Liferay has a compatibility layer that still allows WARs to be deployed and automatically converts them to modules.
- This compatibility layer is extensive and will work out-of-the-box for most portlets.
- While extensive, the compatibility layer does not entirely replicate the legacy behavior of deploying plugins to your appserver.

2.12.16 How Does the Compatibility Layer Differ?

- The goal of the compatibility layer is to support application server independence and ultimately allow for greater modularity.
- To achieve this, some of the configuration common to traditional web-bapps is abstracted away.
- This means that the compatibility layer won't provide the same level of control as a module will.

2.12.17 Traditional WAR Deployment

(Figure 2.29, page 100)

2.12.18 WAR Deployment in the Module Framework

(Figure 2.30, page 100)

2.12.19 WAR Deployment in the Module Framework

- In the Module Framework's compatibility layer, all web applications are deployed as Web Application Bundles (WABs).
- A WAB is simply a WAR file with a manifest added, telling the Module Framework how to deploy the application. This manifest also allows for headers to be added, which allow support for JSPs, tag library definitions, etc.
- The compatibility layer converts a WAR file to a WAB upon deployment.
- The web extender module of the Module Framework detects and deploys this WAB file.

2.12.20 How Can My Plugins Run on Liferay?

- To get traditional WAR-style applications to run on this new version, all you need to do is:
  - Copy your plugin to the new version of the Plugins SDK.
  - Update legacy APIs that have been converted to modules. A comprehensive list of these can be found here:
    `https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/adapting-to-liferay-7s-api`
  - Remove any generated JAR files from your project's `/lib` folder. These will already exist in the Module Framework and so your plugin doesn't need them.
  - Deploy your WAR file.
- As we explained before, you can use **Breaking API Migration Tool** to complete the two previous points.
- This process is strongly recommended even if you plan on converting your application to modules, as it helps you isolate and fix any breaking API changes to make the transition to modules go more smoothly.

- Other breaking changes can also be found here:

  `https://github.com/liferay/liferay-portal/blob/master/readme/7.0/BREAKING_CHANGES.markdown`

2.12.21 How Do I Convert My Plugins into Modules?

- We're glad you've decided to convert your plugins to modules!
- Here's the best way to do so:
  - Upgrade your existing application using the steps previously mentioned.
  - Determine what modules to create (web, api, service).
  - Define manifest attributes and dependencies.
  - Update any Liferay-specific configuration files.
  - Run Service Builder to generate code for your application's service and API modules.
- More information about this process can be found here:

  `https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/modularizing-legacy-plugins`

2.12.22 Liferay workspace

- In Liferay 7.0, we recommend that you move to our new project structure called Liferay Workspace. It is built on Gradle and provides support for your Plugins SDK-based projects. It will also leverage all the new theme tooling we've built in 7.0 and integrate with Liferay IDE/Studio.
- Your new project will look like this:
  - Project-folder
    * Modules
    * Plugins-sdk
    * Themes

- To know how to create this workspace, please check:

  `https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/creating-a-liferay-workspace`

2.12.23 Upgrading themes

- In Liferay 7.0, we've created a new set of theme tools that frontend developers should be much more familiar with. They are built using *Node.js*, *yo*, and *gulp*. If you have an existing theme in your Plugins SDK, you can migrate them to your new Liferay Workspace.
- To upgrade a single theme: `blade migrateTheme [THEME_NAME]`
- To migrate all themes use: `blade migrateTheme -a`

  Your themes will be moved to the new theme folder. Next we will need to convert the CSS styles from bootstrap 2 to bootstrap 3. We will need to leverage another npm package created by Liferay.
- The next thing you will want to do is rename all your SASS files from *.css to .sass*. In Liferay 7.0, SASS files use the proper extension. The SASS compilers will only compile files with the .scss file extension.
- Install it by running: `npm install -g convert-bootstrap-2-to-3`.
- Then you can update the file by running: `bs3 path/to/file`.

  You can run this against HTML files, CSS files, and SASS files. This tool won't fix everything for you, but it will give you a great headstart. The rest is up to you.

Figure 2.20:

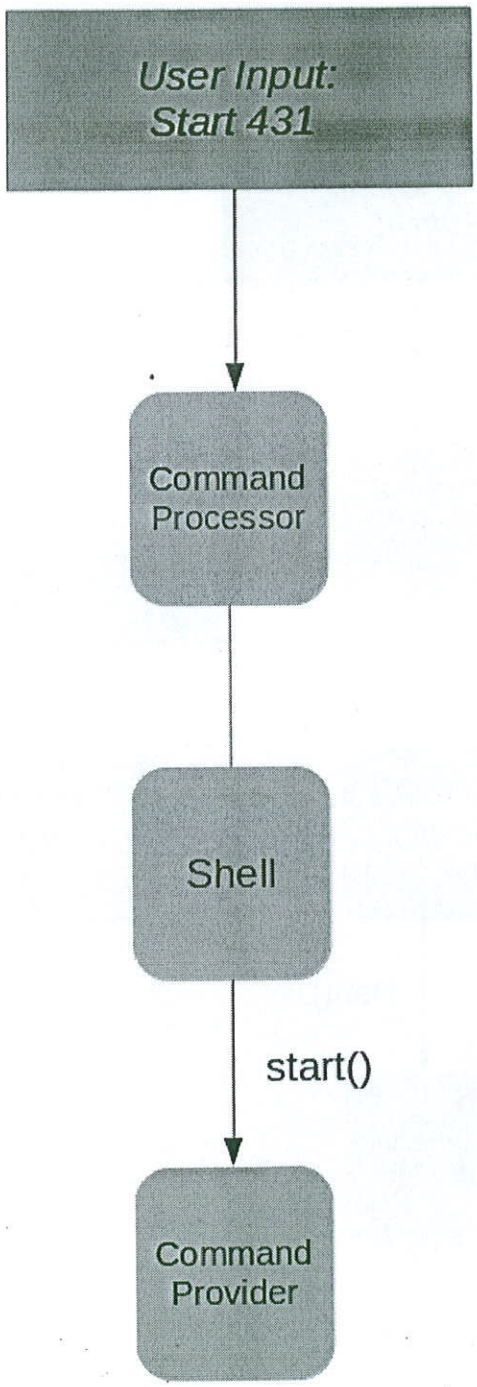


Figure 2.21:

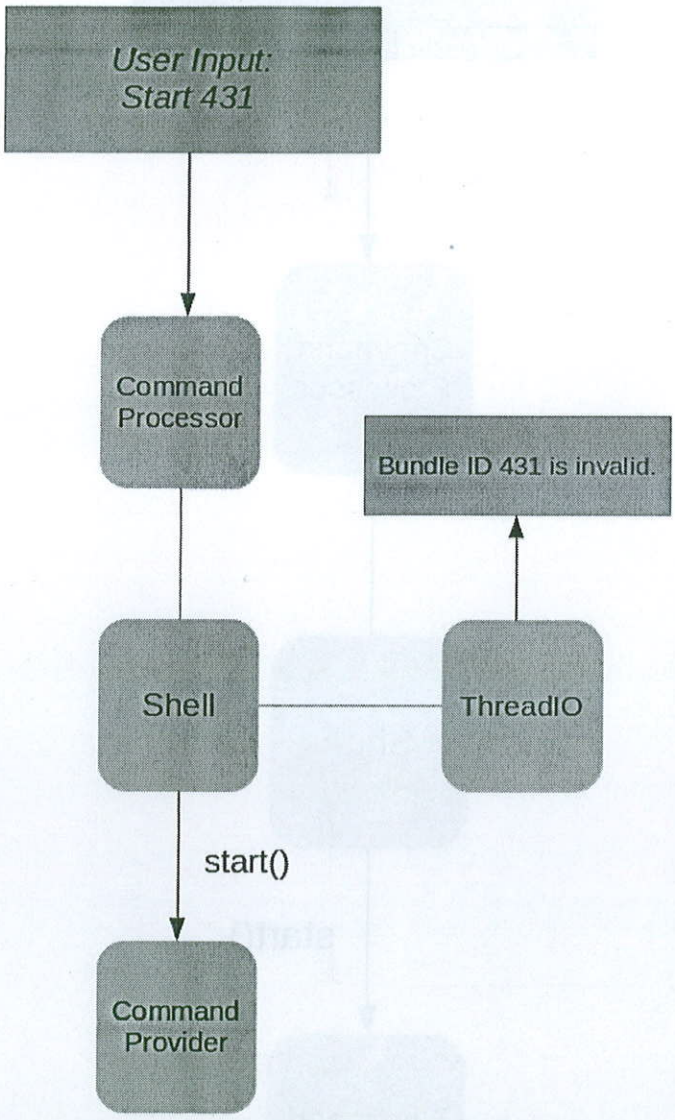


Figure 2.22:

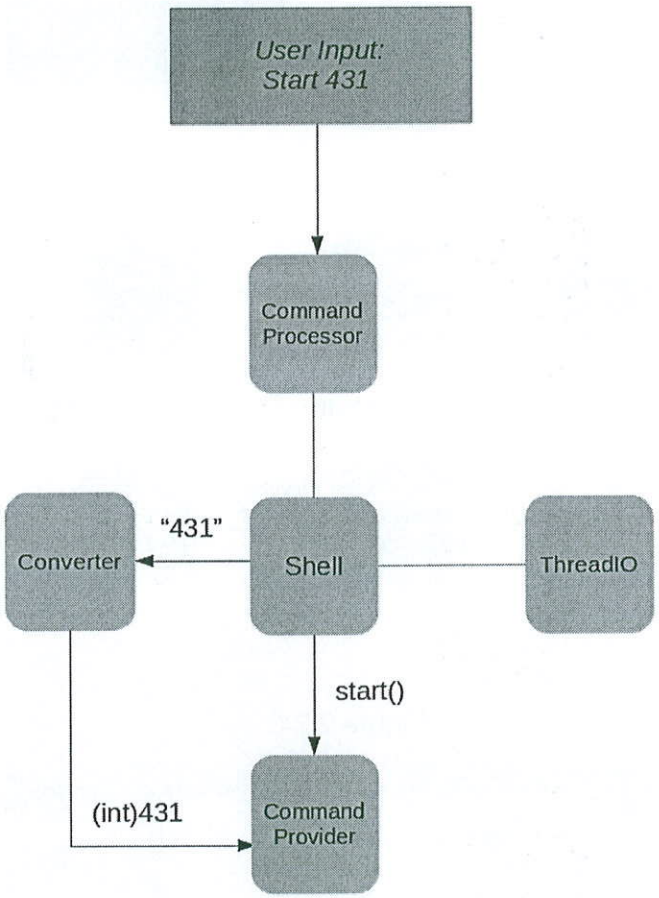


Figure 2.23:

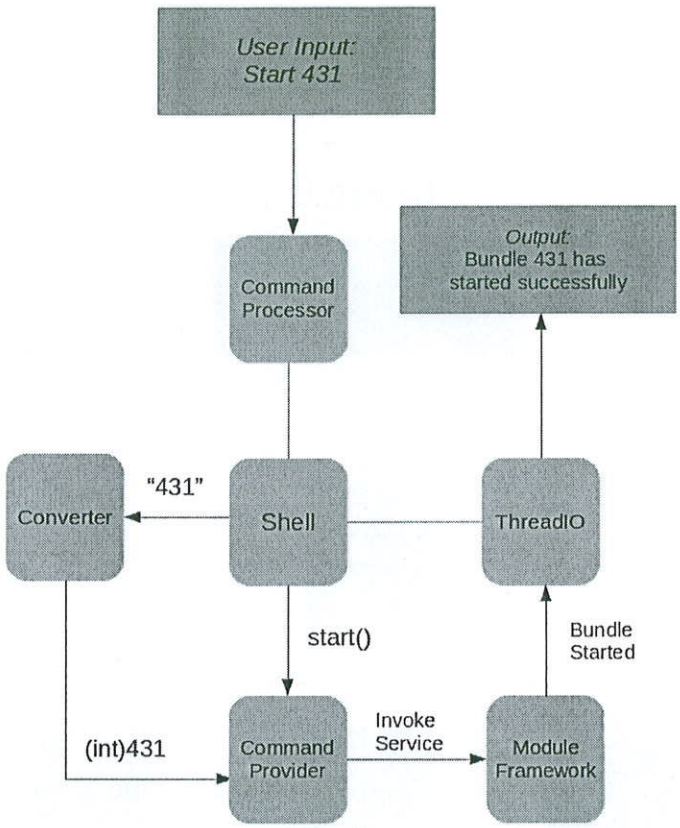


Figure 2.24:

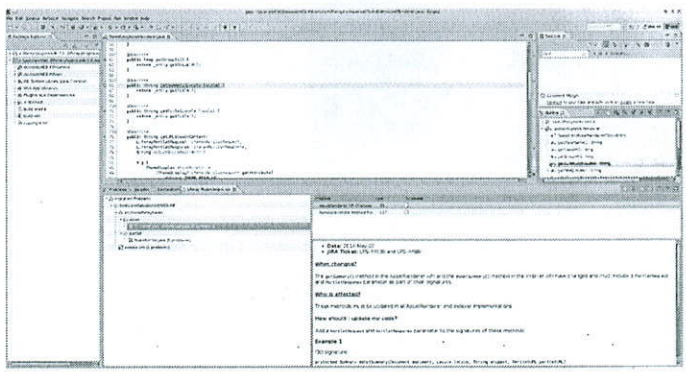


Figure 2.25:

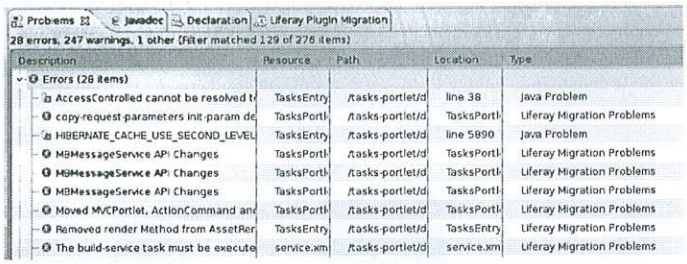


Figure 2.26:

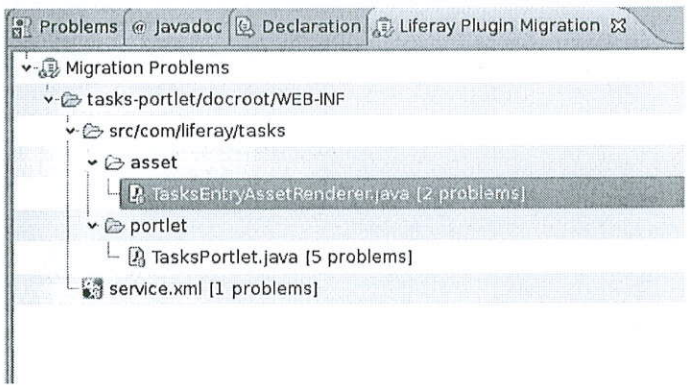


Figure 2.27:

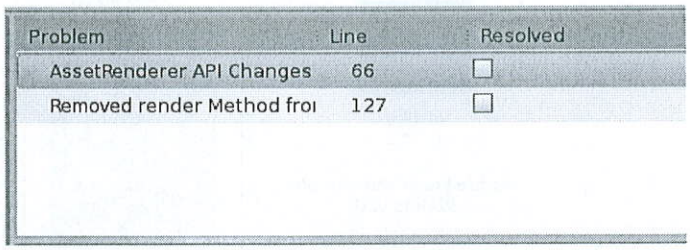


Figure 2.28:

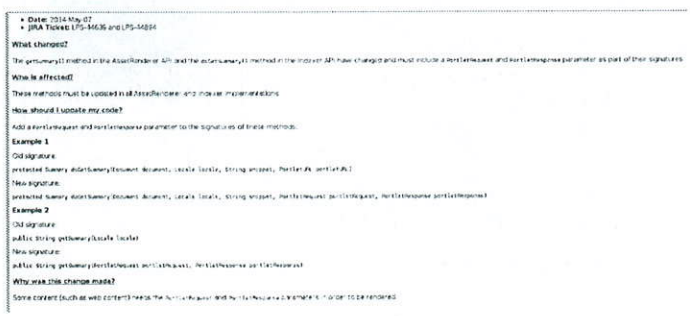


Figure 2.29:

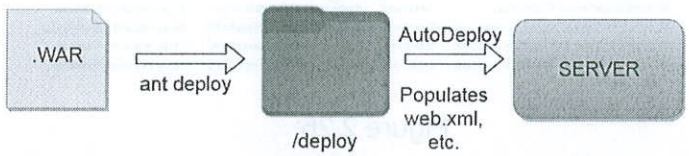


Figure 2.30: