Hands- On Liferay DXP

Learn Portlet Development and Customization Using OSGi Modules

Apoorva Prakash
Shaik Inthiyaz Basha

Apress®

Hands- On Liferay DXP

Learn Portlet Development and Customization Using OSGi Modules

Apoorva Prakash
Shaik Inthiyaz Basha

Apress®

Hands- On Liferay DXP: Learn Portlet Development and Customization Using OSGi Modules

Apoorva Prakash
BANGALORE, India

Shaik Inthiyaz Basha
Nellore, AP, India

ISBN-13 (pbk): 978-1-4842-8562-6
ISBN-13 (electronic): 978-1-4842-8563-3
https://doi.org/10.1007/978-1-4842-8563-3

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Divya Modi
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/us/services/source-code.

Printed on acid-free paper

To my father,
Dr. Om Prakash Srivastava,
to whom I owe everything,
and
who inspired me but is not here to read this.
Yes, life is like that sometimes!
— Apoorva Prakash

Table of Contents

About the Authors

Apoorva Prakash is a Liferay-certified professional who has worked on Liferay for over a decade. Currently, he works with Schneider Electric Pvt Ltd., India, as a Liferay Expert and Engineering Lead for a team working on various projects of different technologies, including NodeJS, Python, AWS-based serverless technologies, and so on. Apoorva has defined the architecture of multiple portals, including large employee portals, ecommerce sites, and so on, in Liferay for over 12 years and counting. His other work areas include NodeJS, Python, AWS, and Kubernetes. Development and deployment are his passions, and he is inherently very keen on attention to detail. He is an avid blogger, and his blog has been mentioned in the Liferay community round-up several times. Apoorva has completed his master's degree in computer application from the school of computer science, Apeejay Institute of Technology, Greater Noida, Uttar Pradesh. His other hobbies are tech blogging and wildlife photography.

Shaik Inthiyaz Basha is a Liferay Architect and Technical Expert at Schneider Electric Pvt Ltd., India. He is an expert in Content Management Systems (CMS) and Amazon Web Service (AWS). Inthiyaz currently holds the position of Platform Architect in a group involved in developing Liferay and Elastic Search applications. His accomplishments in enhancing and creating various Liferay components are evident from his various successful implementations. His experience and knowledge are supported by certificates such as Liferay Backend Developer (DXP). Inthiyaz is also the founder of the https://letuslearnliferay.blogspot.com, which contains a lot of information on Liferay and the CMS world. Since 2011, he has created various kinds of CMS applications, supporting large banking and financial systems. His main area of interest is web applications. Inthiyaz uses Java, AWS, and Elastic Search on a daily basis, but he is open to learning other technologies and solutions. He holds a master's degree in Computer Networks from Quba College of Eng & Tech, Affiliated with JNTUA University, Nellore, Andhra Pradesh, India.

Acknowledgments

In life, one rarely comes across people whose few words or mere presence can bolster you to do something extraordinary. Many people encouraged us and contributed in innumerable ways when writing this book. We want to acknowledge the following key people whose humble support was a constant source of strength during the toil of creating this book:

- Mr. Sanju Varghese Raju
- Senior General Manager, Schneider Electric Pvt Ltd.
- One of the most humble and most genuine person we've met and we thank him for his continuous support, from inception to publishing this book.
- Mr. Veera Vasantha Reddy
- Assistant Vice President, Development Bank of Singapore.

  A technocrat and dear friend, and we thank him for his guidance and critical review comments.
- Our families
- For allowing us to burn the midnight oil and spend weekends on this book.

Introduction

Liferay has been a market leader in ready-to-deploy portals for quite some time. During its lifetime, Liferay has experienced several architectural upgrades that enhanced user and development experiences. Liferay DXP is the most mature version of Liferay. As Liferay matured, it kept adding several technologies; the biggest of which are OSGi and Gradle. OSGi added a layer of modularity to Liferay, whereas Gradle has given the deployment process more flexibility.

This book is a perfect fit for you if you possess basic Java knowledge and are familiar with the Liferay user interface. It's perfect if you want to develop portlet modules in Liferay DXP and customize the default Liferay behavior. You will also learn about OSGi, Blade CLI, the Liferay development environment setup, and best practices. This book will help improve your productivity. If you are hands-on with an older version of Liferay or have little understanding of Liferay's development approach and are looking forward to learning about the nitty-gritty of Liferay—DXP development—this book is a perfect fit for you.

Portlets are the heart and soul of Liferay development, and they can be created using multiple templates such as LiferayMVC, Spring, and others. Portlets are the endpoint for users, from where they can trigger different functionalities, such as database connectivity, IPC, schedulers, and so on.

Liferay is not all about custom development; you can also use its out-of-the-box functionalities to achieve requirements. To utilize its out-of-the-box functionality, you can customize Liferay default behavior in

several ways, including customization of user interfaces, languages, action classes, events, services, and other aspects. Liferay's out-of-the-box search framework can also be used to enable search in custom entities.

We tried to cover all concepts related to hands-on Liferay development and sincerely hope the book fulfills our readers' expectations.

Source Code

All source code used in this book can be downloaded from github.com/Apress/Hands--On-Liferay-DXP-by-Apoorva-Prakash-and-Inthiyaz-Basha.

CHAPTER 1

OSGi Basics

This chapter dives deep into OSGi concepts, along with its essential features, its architecture, services, the Service Registry, and a few other crucial topics that fall under the basics of OSGi concerning Liferay DXP. Further, you learn about bundles with a straightforward example in the next chapters. By the end of this chapter, you will understand the unlimited potential of OSGi.

Understanding OSGi

The Open Services Gateway Initiative (OSGi) was founded in March, 1999 and is managed by the OSGi Alliance.

The OSGi Alliance now refers to the framework specification as OSGi or the OSGi Service Platform. To create a Java-based service framework that can be managed remotely, the vendors of networking providers and embedded systems came together and created a set of standards. OSGi was initially developed to be a gateway for managing Internet-enabled devices like smart appliances. The Java software framework is embedded in a gateway hardware platform, such as a set-top box or cable modem. This software framework acts as a central message dealer to the home's LAN (Local Area Network). The core goal is to efficiently manage cross-dependencies for software developers by creating a standardized middleware for intelligent devices.

A. Prakash and S. I. Basha, *Hands- On Liferay DXP*,
https://doi.org/10.1007/978-1-4842-8563-3_1

Liferay uses OSGi extensively for product development. Other noteworthy companies include Oracle WebLogic, Eclipse Foundation, IBM WebSphere, Atlassian Jira and Confluence, and JBoss. These are the notable companies that are using OSGi for their product development.

Let's look at what makes OSGi different from other frameworks.

How Is OSGi Different?

The OSGi framework is different from other frameworks based on Java, especially Spring. More than one application can exist in the same container in the OSGi bundle runtime environment. The OSGi container takes care of access to the dependencies required by each component in the container. The OSGi framework also supports standardized dependency injection, as defined by the Aries Blueprint project.

In OSGi, bundles can consume services exposed to other bundles. A bundle can define and declare a version of bundles. The runtime will automatically load all its bundles to resolve all dependencies.

Note In OSGi, if any bundle dependencies require multiple versions of the same bundle, they are also available side by side.

OSGi is a modularity layer for the Java platform. OSGi's core specifications define a component and service model for Java. OSGi provides a service-oriented development model, allowing for a service-oriented architecture within a virtual machine.

For example, large Java applications can be challenging to deploy and manage. In order to update deployment, the system/servers need to be cycled, and the application build and deployment may cause system outages. But OSGi provides an isolated module cycling/updating capability to increase availability.

A Deeper Look at OSGi

OSGi offers an elegant solution for handling dependencies, by requiring dependency declarations within units of modularity. Multiple applications can coexist in the same container in OSGi, and the OSGi bundle runtime environment manages them. The OSGi container will ensure each component is sufficiently isolated and approach any dependencies required to access.

OSGi has two parts. The first part is called the *bundle*. It has modular component specifications, which are generally referred to as *plugins*. The specification will help determine the bundle's interaction and lifecycle infrastructure. The second part is the *Service Registry,* which is beneficial to understanding how bundles discover, publish, and bind to services in the Service Oriented Architecture (SOA) at the Java Virtual Machine (JVM) level.

OSGi Architecture

The OSGi architecture is used in different Java-based applications. It is illustrated in Figure 1-1 and consists of several layers that work on top of the hardware and operating system:

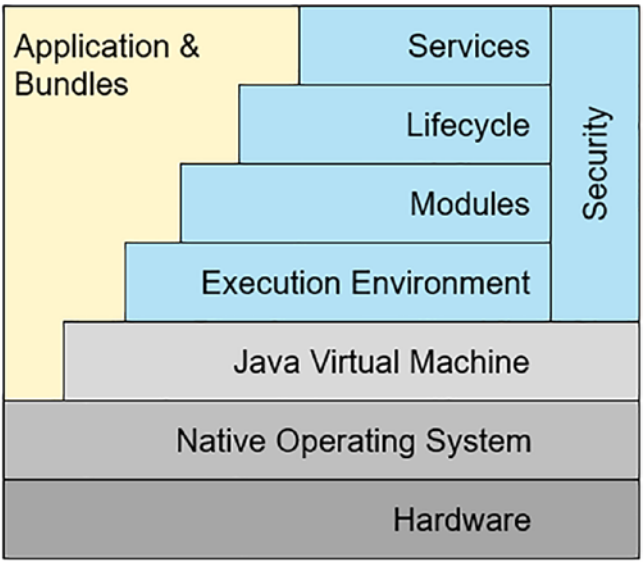

Figure 1-1. *OSGI service gateway*

- **Bundles**: An OSGi bundle is a Java ARchive (JAR) file that contains resources, Java code, and a manifest that describes the bundle and its dependencies. For an application, the OSGi bundle is the unit of deployment. You learn more about bundles in the next section of the chapter.
- **Services**: The services layer in the OSGi architecture will offer a "publish find bind" model for old Java objects to connect bundles dynamically. To simplify, an OSGi service is a Java object instance registered to an OSGi framework with a set of properties.
- **Lifecycle**: The lifecycle layer provides the API used to start, install, uninstall, update, and stop objects.

- **Modules**: This layer defines how to export and import code by bundle.
- **Security**: This layer handles the security aspects.
- **Execution Environment**: This layer defines the available classes and methods in a specific platform.

Now that you've learned a bit about the OSGi basics, the next section explores OSGi bundles.

OSGi Bundles

Bundles are modular Java components. Creating and managing bundles is facilitated by OSGi, and bundles can be deployed in a container. A developer uses OSGi specifications and tools to create one or more bundles. The bundle's lifecycle is defined and managed by OSGi. It also supports the bundles' interactions and hosts them in a container. The OSGi container is roughly parallel to a JVM. Similarly, bundles can be treated as Java applications with distinctive abilities. OSGi bundles run as client and server components inside the OSGi container.

So, bundles are nothing more than OSGi components, and they are in the form of standard JAR files. The only difference between regular JAR files and bundles is the *manifest header* (also referred to as *bundle identifier*s). These manifest headers tell the runtime that this JAR is not a standard JAR file but an OSGi bundle. These bundle identifiers consist of two main parts—`Bundle-SymbolicName` and `Bundle-Version`. You must use a combination of these two to export and import the services. This combination of `Bundle-SymbolicName` and `Bundle-Version` (semantic versioning) creates a unique identifier for OSGi bundles and thus for dependencies.

Note Logically, a bundle has an independent lifecycle with a piece of functionality. It can work independently with start, stop, and remove.

Technically, a bundle is a JAR file containing some OSGi-specific headers in the `MANIFEST.MF` file.

As depicted here, the `Bundle-SymbolicName` is `com.handsonliferay.employee.portlet` and the `Bundle-Version` is `1.2.3.2022`.

```
Bundle-Name: handsonliferay-employee-portlet
Bundle-SymbolicName: com.handsonliferay.employee.portlet
Bundle-Version: 1.2.3.2022
Export-Package: com.handsonliferay.employee
```

Let's deep dive into understanding them:

- `Bundle-SymbolicName`: A unique identifier that refers to the bundle. This is generally understandable human text so that developers can understand the functionality written inside it. The best practice is to name it as class packages are named. In the previous example, you created an API with the symbolic name `com.handsonliferay.employee.portlet`, which any OSGi bundle can import to consume the exposed services. This naming convention is a standard Liferay development approach.
- `Bundle-Version`: 1.2.3.2022. There are four parts to a version number, each separated by three dots (see Figure 1-2). This versioning scheme is also known as *bundle semantic versioning*. Let's look at part of the semantic versioning process:

- *Major*, which is 1 in this case. The major version is changed when there are code changes that can break the code used in the APIs of this bundle.
- *Minor*, which is 2 in this example. This means there are API changes in the bundle, which may include some fixes. It refers to APIs of this bundle and it will not break the code.
- *Micro*, which is 3 is this example. This changes when there are minor changes and no compatibility issues.
- *Qualifier*, which is used when there is no impact to compatibility. It's used to tag snapshots or nightly builds.

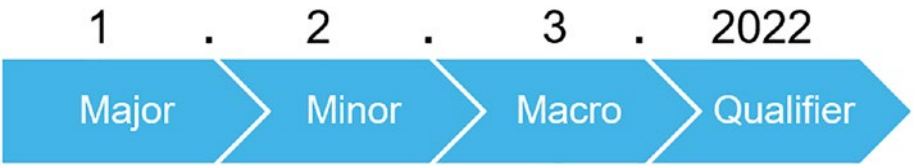

Figure 1-2. *Bundle version*

OSGi Bundle Rules

With these details, you can now learn how OSGI bundles are supposed to work in real-time. Let's look at the OSGI bundle rules:

1. You want code that may include or exclude some application configuration. Using this approach, you will get the primary benefit of modularity.
2. You want code that should update independently from other code. Using this approach, you get another primary benefit of modularity.

3. You want code that has a set of specific dependencies on other libraries. This way, you decrease the chance of conflicts by isolating those dependencies.
4. Some of the interface code might have different implementations. This way, without making any other changes, you can swap out implementations.

Importing and Exporting Bundles

The following example exports a service called `com.handsonliferay.employee.api` with version 1.0.0. The Employee API is exposed with a `Symbolic-Name` of `com.handsonliferay.employee.api` and a version number of `1.0.0`:

```
Bundle-Name: Employee-api
Bundle-SymbolicName: com.handsonliferay.api
Bundle-Version: 1.0.0
Export-Package: com.handsonliferay.api; version=1.0.0
```

Importing Bundles

To understand bundle importing, you must understand version ranges. As you have already learned, an OSGi bundle can be exported with a specific version number, so multiple OSGi bundles have the same symbolic name but different versions. They are essentially nothing but different versions of the same OSGi bundle. There may be cases when you have more than one valid bundle version from various deployed OSGi bundles. To solve this scenario, you can mention ranges in the `import` statement. Square brackets and parentheses are used for this purpose. Square brackets denote inclusiveness, whereas parentheses indicate exclusiveness. You can see this with the following example:

`[2.1, 3.0)` means include version 2.1 up to, but not including, 3.0.

```
Import-Package: com.handsonliferay.employee;
version="[2.1,3.0)"
```

This section has explained the basics of OSGi bundles; in the next section, you explore the OSGi bundle lifecycle.

OSGi Bundle Lifecycle

OSGi is a very dynamic platform, and bundles are the core of this mechanism. A bundle is a state-aware unit, meaning a bundle has several states that it can traverse through and know what state it is in. In traditional OSGi, you have a `BundleActivator`, where you have `start()` and `stop()` methods that are invoked upon the start and stop of the bundle, respectively.

Note Activators are nothing but classes that implement the `org.osgi.framework.BundleActivator` interface.

The OSGi bundle lifecycle layer puts on bundles that can be dynamically started, installed, updated, stopped, and uninstalled. These bundles depend on the module layer for class loading, but it will add an API to manage modules at runtime. See Listing 1-1.

Listing 1-1. Bundle Activator Class

```
package com.handsonliferay.employee.osgi;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
```

```
public class Activator implements BundleActivator{

        @Override
        public void start(BundleContext context) throws
        Exception {
                System.out.println("Starting Hands On
                Liferay");
        }

        @Override
        public void stop(BundleContext context) throws
        Exception {
        System.out.println("Stopping Hands on Liferay");

        }

}
```

Bundle States

Now that you understand that the bundle is state-aware and uses the start and stop methods of a `BundleActivator`, it's time to look at all the possible states that the OSGi bundle can traverse through (see Figure 1-3).

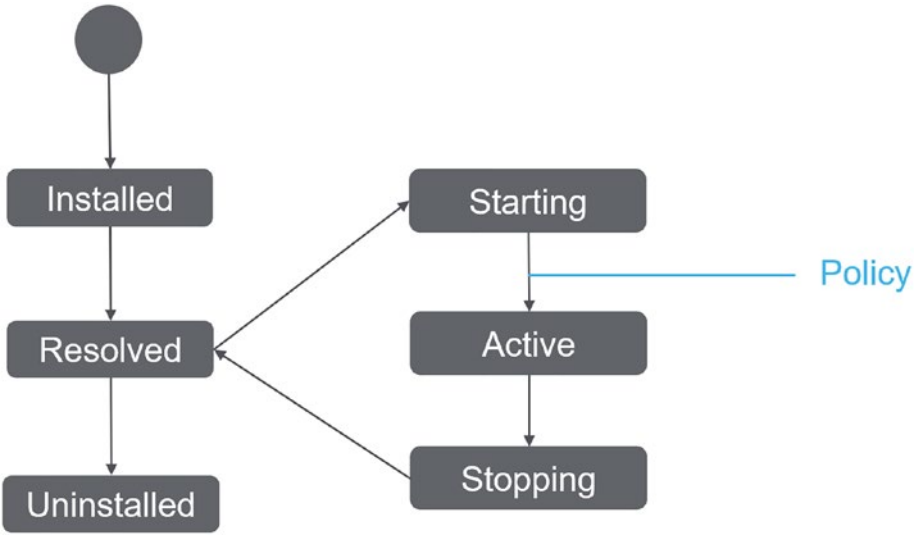

Figure 1-3. *Bundle states*

- **Installed**: This state depicts that the bundle has entered the OSGi runtime. Nothing else; it's not available; running or resolving the dependencies means a bundle is available in the OSGi runtime. If a bundle stays in this state for a long time, that means that it's waiting for some of the bundle's dependencies to be met.
- **Resolved**: This state shows that the bundle has been installed successfully, and the OSGi runtime resolves all the dependencies with the help of the installed bundles in the OSGi runtime. The bundle is available for the next stage, which is Start. Sometimes, the runtime will skip this state if a bundle is started by getting all the required dependencies.

- **Starting**: At this state, the entry-level classes are initialized, and it's a temporary state that the bundle goes through while it is starting and once all dependencies are met.
- **Active**: This state shows that the bundle is up and running.
- **Stopping**: This is a temporary state similar to starting; it goes through this when the bundle stops. All the destructors are called during this state.
- **Uninstalled**: This states shows that the bundle has been removed successfully from the OSGi container.

A bundle lifecycle state will be managed, meaning a bundle can change its state by itself upon deployment, and developers and administrators can manage its state. There are various GUI and command-line tools available to do this. Gogo shell and Apache Felix are two of the most popular tools. You'll see these tools in detail in later chapters.

This section has explained the OSGi bundle lifecycle; in the next section, you explore the OSGi components.

OSGi Components

Any Java class inside a bundle can be declared a component. This can be achieved with the help of declarative services (DS), which provide a service component model on top of the OSGi services. A component can publish itself as a service and make itself available to other components. Similarly, it can consume services published by already installed components.

OSGi components have an independent lifecycle and are reusable, which means you can stop them and start them again without reinstalling them. They will traverse through their lifecycle events again and again. They can have properties and activation policies. An OSGi bundle can have lifecycle methods for activation, deactivation, and configuration.

DS service components are marked with the `@Component` annotation; they implement or extend a service class. These service components can refer to and use each other's services. The Service Component Runtime (SCR) registers component services and handles them by binding them to other components that reference them.

There are two parts to this process—service registration and service handling.

- **Service registration**: When a module containing a service component is installed, the SCR creates a config, binds it with the specified service type, and makes a reference in the Service Registry.
- **Service handling**: When a module referencing a service exposed from another module is installed, SCR searches the Service Registry for a component whose configuration matches the required service type. Once the component is found, SCR binds an instance of that service to the referring member.

Note To understand in a nutshell, you can say—when a module with an exposed service is deployed, SCR registers it in the Service Registry and when a module importing a service is deployed, SCR searches for it in the Service Registry and returns its instance.

`@Component` annotation is a declaration to make the class an OSGi component. `@Referance` annotation marks a field to be injected with a service, and once the Service Registry finds the essential service, it is injected with the resolved service. It can only be used in a `@Component` class.