Resolving Third Party Library Package Dependencies

This document has been updated and ported to Liferay Learn and is no longer maintained here.

Liferay's OSGi framework lets you build applications composed of multiple OSGi bundles (modules). For the framework to assemble the modules into a working system, the modules must resolve their Java package dependencies. In a perfect world, every Java library would be an OSGi module, but many libraries aren't. So how do you resolve the packages your project needs from non-OSGi third party libraries?

Here is the main workflow for resolving third party Java library packages:

Option 1 - Find an OSGi module of the library: Projects, such as Eclipse Orbit and ServiceMix Bundles, convert hundreds of traditional Java libraries to OSGi modules. Their artifacts are available at these locations:

- Eclipse Orbit downloads (select a build)
- ServiceMix Bundles

Deploying the module to Liferay's OSGi framework lets you share it on the system. If you find a module for the library you need, deploy it. Then add a compile-only dependency for it in your project. When you deploy your project, the OSGi framework wires the dependency module to your project's module or web application bundle (WAB). If you don't find an OSGi module based on the Java library, follow Option 2.

Tip: Refrain from embedding library JARs that provide the same packages that Liferay DXP or existing modules provide already.

Note: If you're developing a WAR that requires a different version of a third-party package that Liferay DXP or another module exports, specify that package in your `Import-Package:` list. If the package provider is an OSGi module, publish its exported packages by deploying that module. Otherwise, rename the third-party library (not an OSGi module) differently from the JAR that the WAB generator excludes and embed the JAR in your project.

shows you how to do these things.

Note: Features for manipulating library packages are only available to module projects that use bnd and the `com.liferay.plugin` plugin, such as Liferay Workspace modules. WAR projects must embed libraries wholesale into their classpath.

Note: Liferay's Gradle plugin `com.liferay.plugin` automates several third party library configuration steps. The plugin is automatically applied to Liferay Workspace Gradle module projects created using Liferay Dev Studio DXP or Liferay Blade CLI.

To leverage the `com.liferay.plugin` plugin outside of Liferay Workspace, add code like the listing below to your Gradle project and update the version of the `com.liferay.gradle.plugins` artifact to the latest version found in the repository:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins", version: "4.0.
    }

    repositories {
        maven {
            url "https://repository.liferay.com/nexus/content/repositories/liferay-public-
        }
    }
}

apply plugin: "com.liferay.plugin"
```

If you use Gradle without the `com.liferay.plugin` plugin, you must embed the third party libraries wholesale.

The recommended package resolution workflow is next.

Library Package Resolution Workflow

your module from incorporating unneeded packages.

Here's a configuration workflow for module projects that minimizes dependencies and Java package imports:

1. Add the library as a compile-only dependency (e.g., `compileOnly` in Gradle, `<scope>provided</scope>` in Maven).

2. Copy only the library packages you need by specifying them in a conditional package instruction (`Conditional-Package`) in your `bnd.bnd` file. Here are some examples:

   `Conditional-Package: foo.common*` adds packages your module uses such as `foo.common` , `foo.common-messages` , `foo.common-web` to your module's class path.

   `Conditional-Package: foo.bar.*` adds packages your module uses such as `foo.bar` and all its sub-packages (e.g., `foo.bar.baz` , `foo.bar.biz` , etc.) to your module's class path.

   Deploy your project. If a class your module needs or class its dependencies need isn't found, go back to main workflow **Step 1 - Find an OSGi module version of the library** to resolve it.

   Important: Resolving packages by using compile-only dependencies and conditional package instructions assures you use only the packages you need and avoids unnecessary transitive dependencies. It's recommended to use the steps up to this point, as much as possible, to resolve required packages.

3. If a library package you depend on requires non-class files (e.g., DLLs, descriptors) from the library, then you might need to embed the library wholesale in your module. This adds the entire library to your module's classpath.

Next you'll learn how to embed libraries in your module project.

Embedding Libraries in a Project

You can use Gradle or Maven to embed libraries in your project. Below are examples for adding Apache Shiro using both build utilities.

EMBEDDING LIBRARIES USING GRADLE

```
dependencies {
    compileInclude group: 'org.apache.shiro', name: 'shiro-core', version: '1.1.0'
}
```

The `com.liferay.plugin` plugin's `compileInclude` configuration is transitive. The `compileInclude` configuration embeds the artifact and all its dependencies in a `lib` folder in the module's JAR. Also, it adds the artifact JARs to the module's `Bundle-ClassPath` manifest header.

Note: The `compileInclude` configuration does not download transitive optional dependencies. If your module requires such artifacts, add them as you would another third party library.

Note: If the library you've added as a dependency in your `build.gradle` file has transitive dependencies, you can reference them by name in an `-includeresource:` instruction without having to add them explicitly to the dependency list. See how it's used in the Maven section next.

EMBEDDING A LIBRARY USING MAVEN

Follow these steps:

1. Open your project's `pom.xml` file and add the library as a dependency in the `provided` scope:

    ```
    <dependency>
      <groupId>org.apache.shiro</groupId>
      <artifactId>shiro-core</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
    ```

2. Open your module's `bnd.bnd` file and add the library to an `-includeresource` instruction:

    ```
    -includeresource: META-INF/lib/shiro-core.jar=shiro-core-[0-9]*.jar;lib:=true
    ```

expression `[0-9]*` helps the build tool match the library version to make available on the module's class path. The `lib:=true` directive adds the embedded JAR to the module's class path via the `Bundle-Classpath` manifest header.

Lastly, if after embedding a library you get unresolved imports when trying to deploy to Liferay, you might need to blacklist some imports:

```
Import-Package:\
    !foo.bar.baz,\
    *
```

The `*` character represents all packages that the module refers to explicitly. Bnd detects the referenced packages.

Congratulations! Resolving all of your module's package dependencies, especially those from traditional Java libraries, is a quite an accomplishment.

Related Topics

Importing Packages

Exporting Packages

Creating a Project

« Specifying Dependencies Understanding Excluded JARs »

Liferay

Resources

Downloads

Documentation

Activation Keys

LESA

Subscription

Contact Us

Account Support

Support FAQs

Related Sites

Liferay.com

Marketplace

English (US)