< ALL GUIDES

Creating a Multi Module Project

This guide shows you how to create a multi-module project with Spring Boot. The project will have a library jar and a main application that uses the library. You could also use it to see how to build a library (that is, a jar file that is not an application) on its own.

What You Will Build

You will set up a library jar that exposes a service for simple “Hello, World” messages and then include the service in a web application that uses the library as a dependency.

What You Need

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Create a Root Project.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-multi-module.git`
- cd into `gs-multi-module/initial`
- Jump ahead to Create the Library Project.

When you finish, you can check your results against the code in `gs-multi-module/complete`.

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with Gradle and Maven is included here. If you’re not familiar with either, refer to Building Java Projects with Gradle or Building Java Projects with Maven.

Create a Root Project

This guide walks through building two projects, one of which is a dependency to the other. Consequently, you need to create two child projects under a root project. But first, create the build configuration at the top level. For Maven you will want a `pom.xml` with `<modules>` listing the subdirectories:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.or
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.springframework</groupId>
    <artifactId>gs-multi-module</artifactId>
    <version>0.1.0</version>
    <packaging>pom</packaging>

    <modules>
        <module>library</module>
        <module>application</module>
    </modules>

</project>
```

For Gradle, you will want a `settings.gradle` including the same directories:

```
rootProject.name = 'gs-multi-module'

include 'library'
include 'application'
```

and (optionally) you could include an empty `build.gradle` (to help IDEs identify the root directory).

Create the Directory Structure

In the directory that you want to be your root directory, create the following subdirectory structure (for example, with `mkdir library application` on *nix systems):

```
└── library
└── application
```

In the root of the project, you will need to set up a build system, and this guide shows you how to use Maven or Gradle.

Create the Library Project

One of the two projects serves as a library that the other project (the application) will use.

Create the Directory Structure

In a the `library` directory, create the following subdirectory structure (for example, by using `mkdir -p src/main/java/com/example/multimodule/service` on *nix systems):

```
└── src
    └── main
        └── java
            └── com
                └── example
                    └── multimodule
                        └── service
```

Now you need to configure a build tool (Maven or Gradle). In both cases, note that the Spring Boot plugin is **not** used in the library project at all. The main function of the plugin is to create an executable "über-jar", which we neither need nor want for a library.

Although the Spring Boot Maven plugin is not being used, you do want to take advantage of Spring Boot dependency management, so that is configured by using the `spring-boot-starter-parent` from Spring Boot as a parent project. An alternative would be to import the dependency management as a Bill of Materials (BOM) in the `<dependencyManagement/>` section of the `pom.xml` file.

Setting up the Library Project

For the Library project, you need not add dependencies. The basic `spring-boot-starter` dependency provides everything you need.

You can get a Maven build file with the necessary dependencies directly from the Spring Initializr. The following listing shows the `pom.xml` file that is created when you choose Maven:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apach
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.7.1</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>multi-module-library-initial</artifactId>
```

```
	<version>0.0.1-SNAPSHOT</version>
	<name>multi-module-library-initial</name>
	<description>Demo project for Spring Boot</description>

	<properties>
		<java.version>1.8</java.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

You can get a Gradle build file with the necessary dependencies directly from the Spring Initializr. The following listing shows the `build.gradle` file that is created when you choose Gradle:

```
plugins {
	id 'org.springframework.boot' version '2.7.1'
	id 'io.spring.dependency-management' version '1.0.11.RELEASE'
	id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
	mavenCentral()
}

dependencies {
```

```
        implementation 'org.springframework.boot:spring-boot-starter'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Adjusting the Library Project

If you generated the Library project from `start.spring.io` it will contain a wrapper script for the build system (`mvnw` or `gradlew` depending on the choice you made). You can move that script and its associated configuration up to the root directory:

```
$ mv mvnw* .mvn ..
$ mv gradlew* gradle ..
```

The Library project has no class with a main method (because it is not an application). Consequently, you have to tell the build system to not try to build an executable jar for the Library project. (By default, the Spring Initializr builds executable projects.)

To tell Maven to not build an executable jar for the Library project, you must remove the following block from the `pom.xml` created by the Spring Initializr:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

The following listing shows the final `pom.xml` file for the Library project:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apach
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.7.1</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>library-complete</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>multi-module-library-complete</name>
        <description>Demo project for Spring Boot</description>
```

```
	<properties>
		<java.version>1.8</java.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>

</project>
```

To tell Gradle to not build an executable jar for the Library project, you must add the following blocks to the `build.gradle` created by the Spring Initializr:

COPY

```
plugins {
  id 'org.springframework.boot' version '2.5.2' apply false
  id 'io.spring.dependency-management' version '1.0.11.RELEASE'
  // ... other plugins
}

dependencyManagement {
  imports {
    mavenBom
org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES
  }
}
```

The `bootJar` task tries to create an executable jar, and that requires a `main()` method. As a result, you need to disable it by disabling the the Spring Boot plugin, while keeping it for its dependency management features.

The following listing shows the final `build.gradle` file for the Library project:

```
plugins {
	id 'org.springframework.boot' version '2.7.1' apply false
	id 'io.spring.dependency-management' version '1.0.11.RELEASE'
	id 'java'
}
```

```
group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
        mavenCentral()
}

dependencyManagement {
        imports {
                mavenBom org.springframework.boot.gradle.plugin.SpringBootPlugin
        }
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Create a Service Component

The library will provide a `MyService` class that can be used by applications. The following listing (from `library/src/main/java/com/example/multimodule/service/MyService.java`) shows the `MyService` class:

COPY

```
package com.example.multimodule.service;

import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.stereotype.Service;

@Service
@EnableConfigurationProperties(ServiceProperties.class)
public class MyService {

  private final ServiceProperties serviceProperties;

  public MyService(ServiceProperties serviceProperties) {
    this.serviceProperties = serviceProperties;
  }

  public String message() {
    return this.serviceProperties.getMessage();
  }
}
```

To make it configurable in the standard Spring Boot idiom (with `application.properties`), you can also add a `@ConfigurationProperties` class. The `ServiceProperties` class (from `library/src/main/java/com/example/multimodule/service/ServiceProperties.java`) fills that need:

```
package com.example.multimodule.service;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("service")
public class ServiceProperties {

  /**
   * A message for the service.
   */
  private String message;

  public String getMessage() {
    return message;
  }

  public void setMessage(String message) {
    this.message = message;
  }
}
```

COPY

You need not do it this way. A library might merely provide pure Java APIs and no Spring features. In that case, the application that consumes the library would need to provide the configuration itself.

Testing the Service Component

You will want to write unit tests for your library components. If you provide re-usable Spring configuration as part of the library, you might also want to write an integration test, to make sure that the configuration works. To do that, you can use JUnit and the `@SpringBootTest` annotation. The following listing (from `library/src/test/java/com/example/multimodule/service/MyServiceTest.java`) shows how to do so:

```
package com.example.multimodule.service;

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
```

COPY

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest("service.message=Hello")
public class MyServiceTest {

  @Autowired
  private MyService myService;

  @Test
  public void contextLoads() {
    assertThat(myService.message()).isNotNull();
  }

  @SpringBootApplication
  static class TestConfiguration {
  }

}
```

In the preceding listing, we have configured the `service.message` for the test by using the default attribute of the `@SpringBootTest` annotation. We do **not** recommend putting `application.properties` in a library, because there might be a clash at runtime with the application that uses the library (only one `application.properties` is ever loaded from the classpath). You **could** put `application.properties` in the test classpath but not include it in the jar (for instance, by placing it in `src/test/resources`).

Create the Application Project

The Application project uses the Library project, which offers a service that other projects can use.

Create the Directory Structure

In the `application` directory, create the following subdirectory structure (for example, with `mkdir -p src/main/java/com/example/multimodule/application` on *nix systems):

```
└── src
    └── main
        └── java
            └── com
                └── example
                    └── multimodule
                        └── application
```

Do not use the same package as the library (or a parent of the library package) unless you want to include all Spring components in the library by `@ComponentScan` in the application.

Setting up the Application Project

For the Application project, you need the Spring Web and Spring Boot Actuator dependencies.

You can get a Maven build file with the necessary dependencies directly from the Spring Initializr. The following listing shows the `pom.xml` file that is created when you choose Maven:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apach
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.7.1</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>multi-module-application-initial</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>multi-module-application-initial</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>1.8</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-actuator</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
```

```
                        </plugin>
                </plugins>
        </build>

</project>
```

You can get a Gradle build file with the necessary dependencies directly from the Spring Initializr. The following listing shows the `build.gradle` file that is created when you choose Gradle:

```
plugins {
        id 'org.springframework.boot' version '2.7.1'
        id 'io.spring.dependency-management' version '1.0.11.RELEASE'
        id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
        mavenCentral()
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-actuator'
        implementation 'org.springframework.boot:spring-boot-starter-web'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

You can delete the `mvnw` and/or `gradlew` wrappers and their associated configuration files:

```
$ rm -rf mvnw* .mvn
$ rm -rf gradlew* gradle
```

COPY

Adding the Library Dependency

The Application project needs to have a dependency on the Library project. You need to modify your Application build file accordingly.

For Maven, add the following dependency:

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>library</artifactId>
  <version>${project.version}</version>
</dependency>
```

COPY

The following listing shows the finished `pom.xml` file:

COPY

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.7.1</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>multi-module-application-complete</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>multi-module-application-complete</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>1.8</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-actuator</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>
                <dependency>
                        <groupId>com.example</groupId>
                        <artifactId>library-complete</artifactId>
                        <version>${project.version}</version>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-
plugin</artifactId>
                        </plugin>
```

```
                </plugins>
        </build>

</project>
```

For Gradle, add the following dependency:

```
implementation project(':library')
```

COPY

The following listing shows the finished `build.gradle` file:

```
plugins {
        id 'org.springframework.boot' version '2.7.1'
        id 'io.spring.dependency-management' version '1.0.11.RELEASE'
        id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
        mavenCentral()
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-actuator'
        implementation 'org.springframework.boot:spring-boot-starter-web'
        implementation project(':library')
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

COPY

Write the Application

The main class in the application can be a `@RestController` that uses the `Service` from the library to render a message. The following listing (from `application/src/main/java/com/example/multimodule/application/DemoApplication.java`) shows such a class:

```
package com.example.multimodule.application;

import com.example.multimodule.service.MyService;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

COPY

```
@SpringBootApplication(scanBasePackages = "com.example.multimodule")
@RestController
public class DemoApplication {

  private final MyService myService;

  public DemoApplication(MyService myService) {
    this.myService = myService;
  }

  @GetMapping("/")
  public String home() {
    return myService.message();
  }

  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }
}
```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration`: Tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if `spring-webmvc` is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a `DispatcherServlet`.
- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the `com/example` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there was not a single line of XML? There is no `web.xml` file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

Because `DemoApplication` is inside a different package (`com.example.multimodule.application`) than `MyService` (`com.example.multimodule.service`), `@SpringBootApplication` cannot automatically detect it. There are different ways to let `MyService be picked up:

- Import it directly with `@Import(MyService.class)`.

- Fetch everything from its package by using `@SpringBootApplication(scanBasePackageClasses={…})`.
- Specifying the parent package by name: `com.example.multimodule`. (This guide uses this method)

If your application also uses JPA or Spring Data, the `@EntityScan` and `@EnableJpaRepositories` (and related) annotations inherit only their base package from `@SpringBootApplication` when not explicitly specified. That is, once you specify `scanBasePackageClasses` or `scanBasePackages`, you might also have to also explicitly use `@EntityScan` and `@EnableJpaRepositories` with their package scans explicitly configured.

Create the `application.properties` File

You need to provide a message for the service in the library in `application.properties`. In the source folder, you need to create a file named `src/main/resources/application.properties`. The following listing shows a file that would work:

```
service.message=Hello, World
```

COPY

Test the Application

Test the end-to-end result by starting the application. You can start the application in your IDE or use the command line. Once the application is running, visit the client application in the browser, at `http://localhost:8080/`. There, you should see `Hello, World` reflected in the response.

If you use Gradle, the following command (really two commands run in sequence) will first build the library and then run the application:

```
$ ./gradlew build && ./gradlew :application:bootRun
```

COPY

If you use Maven, the following command (really two commands run in sequence) will first build the library and then run the application:

```
$ ./mvnw install && ./mvnw spring-boot:run -pl application
```

COPY

Summary

Congratulations! You have used Spring Boot to create a re-usable library and then used that library to build an application.

See Also

The following guides may also be helpful:

- Building an Application with Spring Boot
- Centralized Configuration

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.