

Spring Boot Microservices: Getting Started

Contents

Overview	3
The Spring Framework	3
Features of Spring Boot	11
Spring Boot Simplifies Spring	11
Spring vs. Spring Boot – Build System.....	12
Spring vs. Spring Boot - Dependency	13
Spring Boot: Starter Dependencies.....	14
Spring Boot: Simplify pom.xml.....	15
Spring vs. Spring Boot – Auto configuration	15
Spring Boot Auto Configuration	16
Spring vs. Spring Boot – Templates.....	17
Spring vs. Spring Boot – Security	18
Spring Initializr	19
Importing Maven Project	20
1. Generate Maven Project using Maven	20
2. Import Maven Project inside Eclipse	21
Building a Spring Boot Application.....	23
3. Update Dependency Library in POM.xml.....	23
4. Update Main Method in App.java Class.....	25
5. Run to test the Application	26
6. Implement Simpler Annotation in entry method to Run Spring Boot Application.....	28
Running on the Embedded Tomcat Server	29
1. Update Dependency in Pom.xml	29
2. Entry Class Implementing SpringBootApplication Annotation	30
3. Prepare Controller Class	30
4. Run and Test the Stand alone Web Application	31
Running on The Jetty Server	34
1. Update Dependency in Pom.xml	34
2. Implementation of Main Entry point of the Application	35
3. Implementation of Controller Class.....	36
4. Run and Test the Application.....	37

Spring Boot Microservices: Getting Started

Generating and Deploying and External WAR File.....	38
1. Update Dependency in Pom.xml	38
2. Implementation of Main Entry Point of the Application	39
3. Implementation of Controller Class.....	41
4. Run to test the WAR application.....	41
5. Build WAR using Maven	42
6. Deploy War into Tomcat	43
7. Start Tomcat Server	44
8. Test on Browser	45
9. Check and Stop Tomcat Service	45
Using Spring Initializr	47
1. Get Spring Boot Template Project from http://start.spring.io	47
2. Import Maven Project in Eclipse	50
3. Create new Controller Class.....	52
4. Build Maven Project.....	54
5. Run and Test the Application	55
Performing Message Internationalization	56
1. Update Dependency POM XML	56
2. Entry Point Default Entry Point Application (SpringinitializerApplication).....	57
3. Create New Controller Class	57
4. Create new HTML File under src/main/resources/template folder	58
5. Create New Service Class to Handle Internalization	61
6. Create Mode Properties files for another Locale Language	62
Configuring the Server Port	65
Configuring User-defined Properties	68
Summary	75
Quiz	76

Spring Boot Microservices: Getting Started

Overview

Setting up a new Spring Boot application is simple, and with the use of starter templates, the parent POM, and auto-configurations, running a Spring Boot application is also easy. In this course, you'll explore the Spring Boot framework and features of Spring Boot and the basic design principle of inversion of control achieved in Spring using dependency injection.

Next, you'll learn how to run an embedded web server in a simple Spring Boot application, set up a new Spring Boot project using the Spring Initializr and Spring Tools Suite, and configure properties for your Spring Boot application.

The Spring Framework



Spring Boot is a way for Java developers to work with the Spring Framework in an easy way. Now, before you use Spring Boot you need to understand what the Spring Framework is all about. And that's exactly what we'll discuss fairly quickly here in this video. It's assumed that you have some familiarity with Spring, but if you don't, you can get started here. If you've been a Java developer for any length of time, it's quite likely that you've used some component or the other, which is part of the Spring Framework. Because Spring is a Java framework that provides a convenient way of integrating disparate components into a working enterprise-grade application. The Spring Framework relies heavily on inversion of control to bring these components together.

Spring Boot Microservices: Getting Started



Spring

Refers to an entire family of projects that support a wide range of application scenarios, such as

- large enterprise applications,
- simple cloud-hosted servers,
- standalone batch, and
- integration workloads

Managing dependencies within your application is a hard task, especially in Java, when you have a number of libraries that you could depend on to get your task done. Inversion of Control, implemented using dependency injection, makes this easy in Spring. A common scenario which you might have encountered is to use Spring modules such as Spring MVC to build web applications, but Spring is so much more. Spring refers to an entire family of projects which support a wide range of applications. Spring supports a wide range of enterprise-grade applications, whether they are web applications, long running batch processing jobs or any other kind. You can use Spring with simple cloud-hosted servers, standalone batch workloads and integration workloads.



A common factor that you'll encounter across all enterprise-grade applications is their complexity. There are different components which perform different actions, and these components have to interact with one another. These components might be dependent on one another. Managing dependencies is an important part of any application. Handling and managing these dependencies and ensuring that every component has access to all of the classes that it depends on is an important part of application development. Now, dependencies cannot be managed in an ad hoc manner. This will suffice for simple

Spring Boot Microservices: Getting Started

dependencies. But within an application, this will soon get out of control, which means you need some kind of design pattern to manage these dependencies.

Now design patterns are great and they help you build robust applications. However, design patterns are often not sufficient by themselves. Because design patterns tell you what best practices you need to follow, but you still need to implement them. The onus or the responsibility is on the developer to implement these design patterns.

This means if you're working on a large team of developers, it's quite possible that inexperienced members don't follow the pattern perfectly. You need a way to shift the responsibility from developers to the framework, and this is what Spring does so well. Spring uses Inversion of Control, or IoC, implemented using dependency injection to free the developer from the burden of managing dependencies. When you use the Spring Framework, you can configure the components of your application using either Java annotations or xml configuration files. A component simply has to declare its dependencies. And the Spring Framework will take care of ensuring that the component has access to the classes that it's dependent.

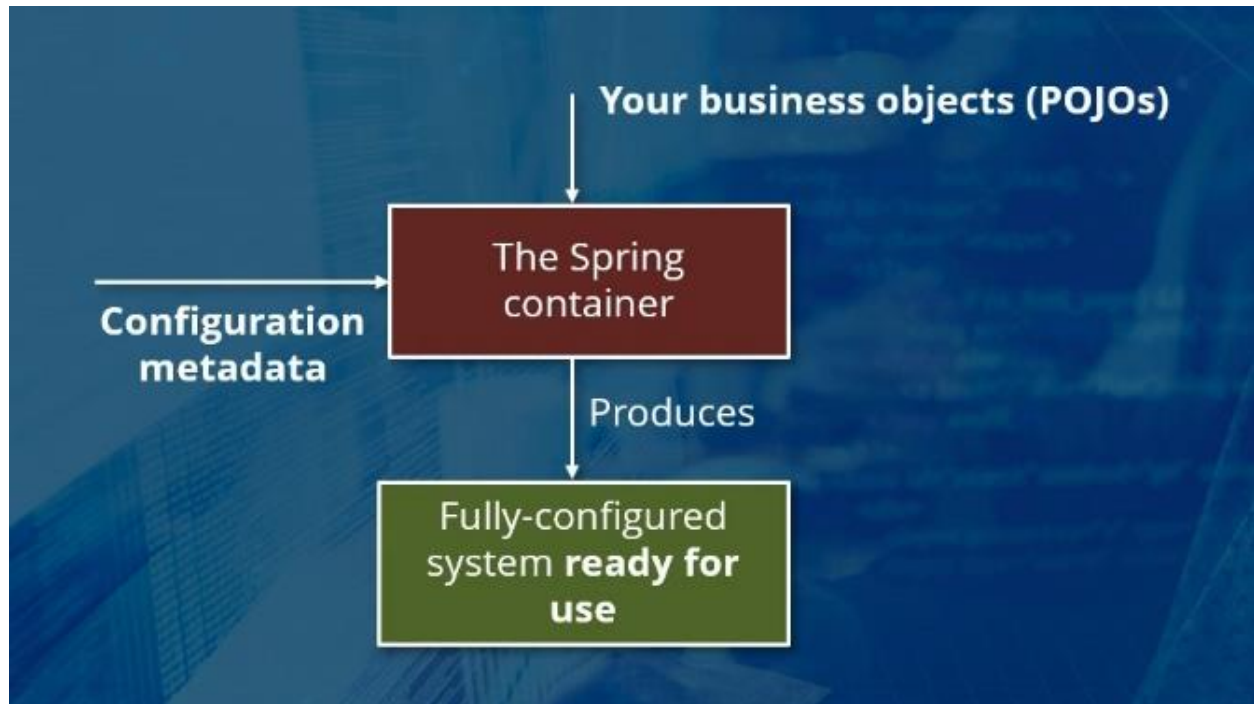


There are a large number of Spring modules available that allow you to work with Spring where you write just simple methods to perform very complex tasks. For example, let's say you need to connect to a database and run queries on this database.

Spring allows you to set up database transactions with simple Java methods without you having to deal with transaction APIs. Spring makes it very easy for you to expose HTTP endpoints from your applications that can respond to HTTP request. These HTTP endpoints can be set up without you having to deal with Servlet APIs. Spring allows you to configure message handlers, which can send messages between the components of your application. And you don't have to deal with the Java Messaging Service APIs directly. And finally, when you work with the Spring Framework, you get monitoring

Spring Boot Microservices: Getting Started

support without dealing directly with JMX APIs. So as you can see, Spring allows you to harness the functionality provided in its different modules in a straightforward manner.



Here is a big picture view of how exactly the Spring Framework works. When you're working with Spring, you're typically writing your code in the form of business objects that are POJOs or plain old Java objects. These POJOs, which are your business objects, will be managed by the Spring container.

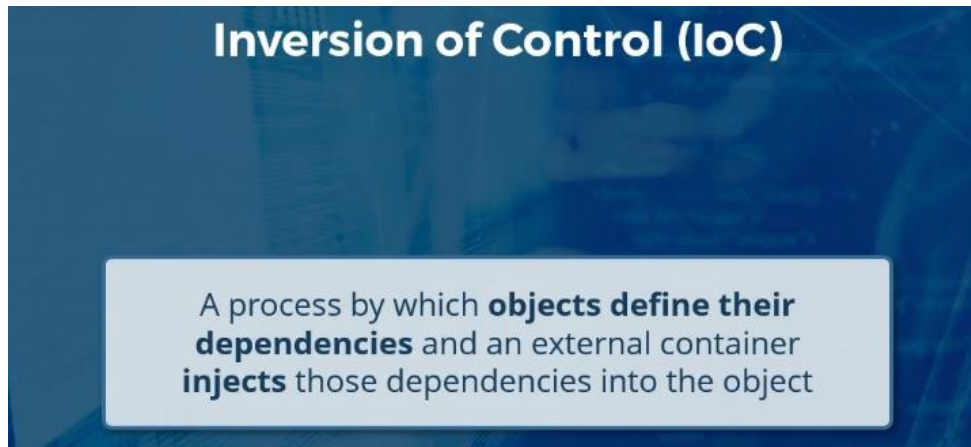
In addition, you might specify additional configuration metadata for your object. This can be via Java annotations or via XML configuration files. This configuration metadata will be applied to your business objects, and in the end, you'll get a fully configured system that is ready for use.



The core fundamental building block of Spring is the Spring Bean. Spring Beans are objects that form the backbone of your Spring application. This can be user-defined beans or beans that are available as a part of the Spring Framework. The basic feature of Spring Beans is the fact that they are completely managed

Spring Boot Microservices: Getting Started

by the Spring Framework. Their entire life cycle, starting from creation, the way they work during their lifetime, and deletion, all of this is handled by Spring.



Spring Beans form a core feature of the dependency management system available in Spring, the Inversion of Control. Inversion of Control is a process by which objects define their dependencies and an external container injects those dependencies into the object.

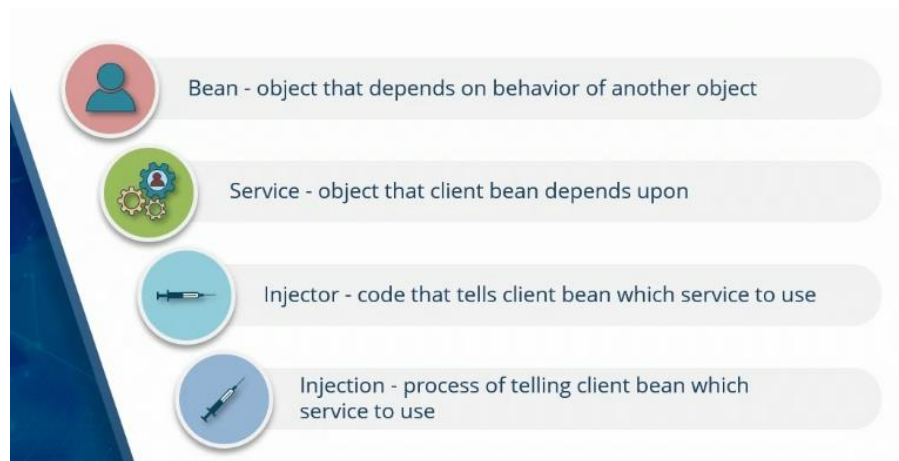


Any Spring Bean within the Spring Framework only has to state what its dependencies are. It's not responsible for creating instances of those objects and external container will take care of this. As a developer, when you develop your own Spring Bean components, you only have to declare what your component is dependent on. You are not responsible for creating or instantiating those dependencies.

Spring Boot Microservices: Getting Started



In order to manage dependencies using dependency injection and the Inversion of Control mechanism, Spring depends on these four components, the Bean, the Service, the Injector and the process of Injection.



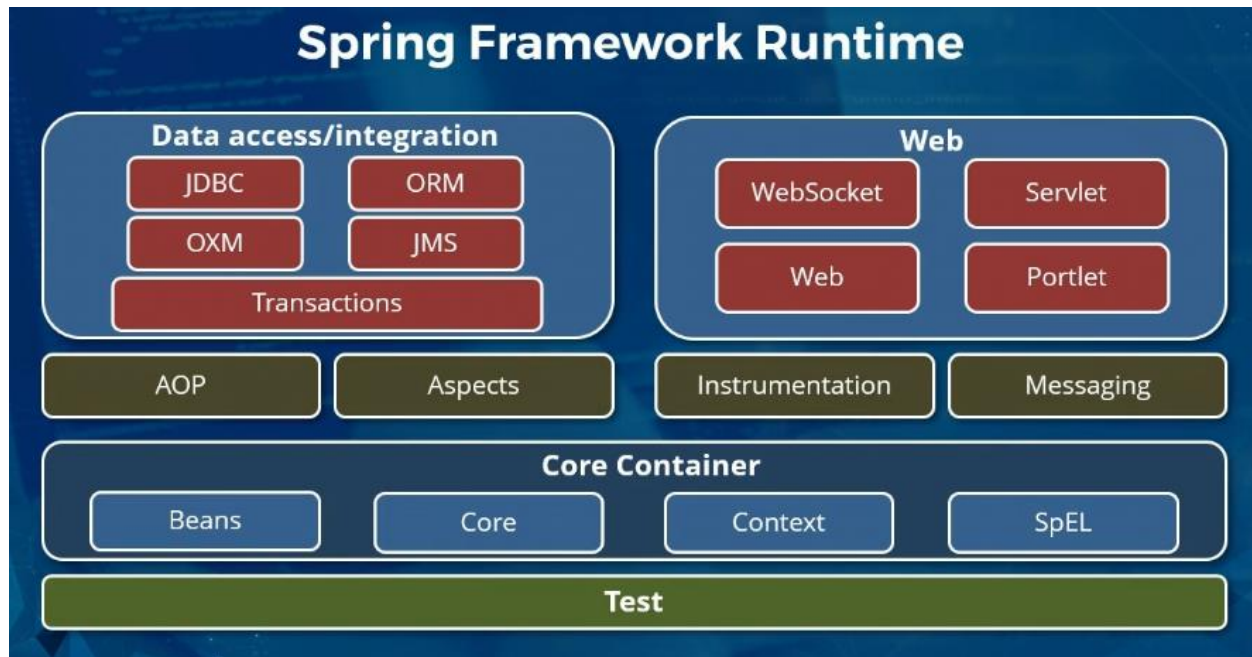
A bean is any object within your application that depends on the behavior of another object. A single bean can have multiple dependencies and other beans can depend on the bean that you create.

The term service typically refers to an object that the client bean depends on. The service is what exhibits the behavior that the client bean uses.

The injector refers to the code that is part of the external Spring container, which tells the client bean which service to use.

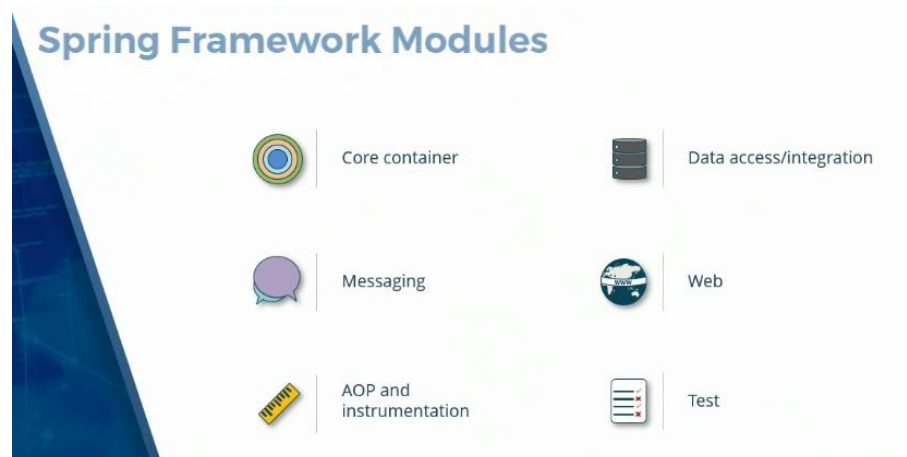
And finally, the way the client bean, a component within your Spring application gets reference to the right service which has the right behavior, is via the process of Injection.

Spring Boot Microservices: Getting Started



Here is a big picture view of the modules included as a part of the Spring Framework runtime. You have data access and integration modules that use JDBC, ORM frameworks such as Hibernate. All of these serve to abstract you away from the transaction mechanics of the underlying database. You have a web related modules as well, which help you build REST services and web applications. You have modules that enable Aspect Oriented Programming, where you can write code to extend the functionality of existing components. You have modules for instrumentation and messaging as well.

At the heart of it all is the Spring Core Container, which takes care of dependency management using dependency injection. There are also a number of testing frameworks that work with Spring, allowing you to test your application end to end.



As a Java developer, anything that you need to develop a fully fledged production-ready application is available in Spring. You simply have to use the right component, set up the right configuration, either in XML or via annotations, and you're good to go. I won't discuss each of the Spring modules in detail. You

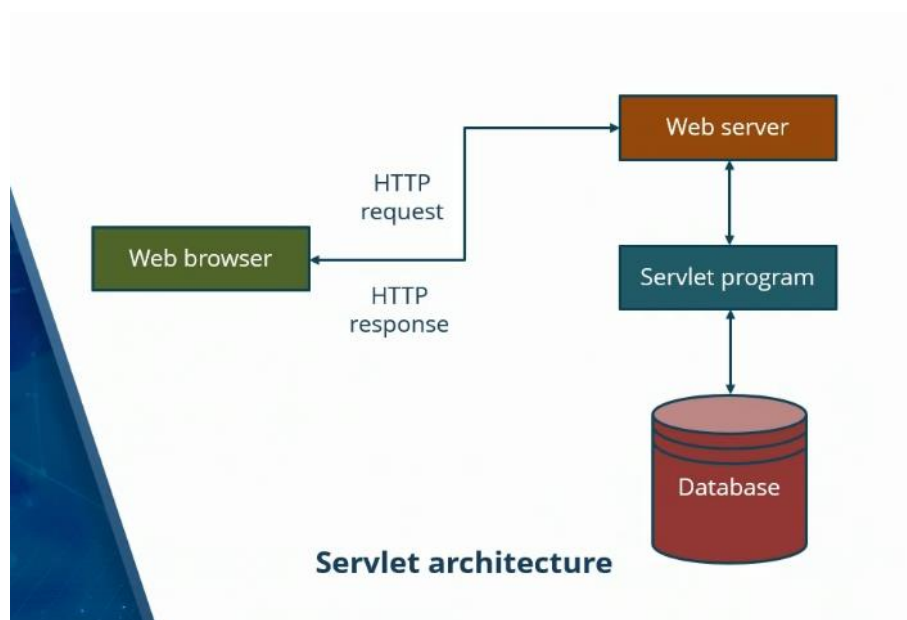
Spring Boot Microservices: Getting Started

will be encountering many of these modules when we actually build our applications using Spring Boot. But let's discuss Spring MVC briefly because we'll use that a lot.



Spring MVC is the framework within Spring that supports the use of the Model-View-Controller design pattern for building web applications.

The model is the representation of the data that you want to show to the user. The view is what is rendered to the user and the controller manages interactions between the model and the view. Spring MVC uses the Servlet architecture, a simple overview of which is present here.



You can see that the Web browser makes HTTP request and gets HTTP responses from the Web server. Within the Web server, Spring MVC has a Servlet program running, which will look at the incoming web request and get the right handler to respond to that web request. The response handler responsible for that web request may process the web request by accessing a Database, querying a Database or updating a Database. Once the handler has finished processing the request, a response is sent back to the user.

Spring Boot Microservices: Getting Started

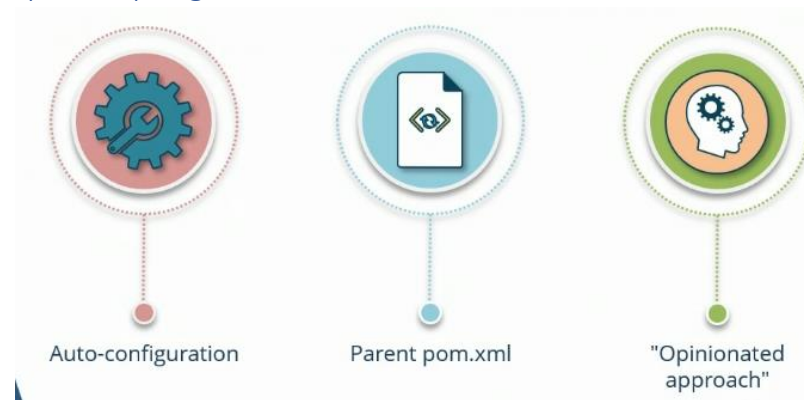
Features of Spring Boot

When you're building a fairly heavyweight application using Spring, you might find it hard to manage all of the dependencies that you need. Spring can be hard to work with when you first start off and this is where Spring Boot really shines. Spring Boot makes it easy for you to work with Spring.



Spring Boot is not an entirely new framework, in fact, it's an extension of the Spring framework. Spring Boot includes all of the same modules that Spring does. The whole objective of working with Spring Boot is to simplify how you develop your Spring-based applications. Spring Boot does this by automatically pre-configuring virtually all of the third party libraries that you will use. It's quite possible that you've developed Spring applications using the Spring framework directly. And this can get hard especially as a number of component that you work with increases. With Spring Boot, you will find that your Spring applications just work. It seems almost like magic.

Spring Boot Simplifies Spring



If you work with Spring directly and you switch to Spring Boot, it's almost magical and hard to believe how easy Spring Boot makes everything. Let's discuss how Spring Boot simplifies Spring.

It uses Auto-configuration. Auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you've added. Based on the dependencies you have in your app, your app will be configured automatically. Spring Boot also makes use of the Parent pom.xml. Instead of explicitly specifying all the dependencies and versions that you need within your pom.xml, in your Maven build configuration, you'll simply use a parent pom which has most of the dependencies

Spring Boot Microservices: Getting Started

pre-configured with the right versions as well. The reason why Spring Boot is so much easier than working with Spring directly is because Spring Boot takes the Opinionated approach to setting up your application. This opinionated approach on the part of Spring Boot is what makes it easy to create standalone production grade Spring-based applications that just work.



What exactly does it mean that Spring has an opinionated approach? In software development, this term refers to an approach that intentionally reduces the flexibility available to the developer by making choices on behalf of the developer. This opinionated approach of Spring Boot basically means that you as a developer have to give up some control. The reason why your Spring Boot applications just work is because Spring Boot automatically provides implementations and sensible defaults. You can choose to go with these defaults if you want to, and things will work out of the box. This reduces the flexibility that you have as a developer, but overall simplifies the development process. It's important to understand though that this opinionated approach doesn't stop you from making granular configuration specifications for your app. You can still configure those things that you want to and use the defaults for the remaining components.

Spring vs. Spring Boot – Build System

<ul style="list-style-type: none">• Spring supports several build systems - Maven, Gradle, Ant, and others	<ul style="list-style-type: none">• Spring Boot recommends Maven or Gradle rather than Ant• Works best with a build system that can pull artifacts published to Maven Central
---	--

I'll give you a few examples of Spring Boot's opinionated approach. Now when you use Spring, Spring supports several build systems. You can use Maven, Gradle, Ant and others, that choice is up to you. When you use Spring Boot, the recommended build systems are Maven or Gradle. Spring Boot doesn't really have great support for the Ant build system. Spring Boot, in fact, works best with a build system that can pull artifacts published to Maven Central. So here is Spring Boot's opinion to get things to just work. Use a build system that works with artifacts in Maven Central, which is why we'll be using Maven for all of our demos.

Spring Boot Microservices: Getting Started

Spring vs. Spring Boot - Dependency



If you set up a standalone Spring application, you know that Spring needs detailed dependencies and corresponding versions to be specified by you, the developer, before you can even get started. With Spring Boot, you can simply use the starter dependencies. These starter dependencies will pull in sensible defaults on your behalf. If you need to change those defaults, you will need to specify additional configuration within your pom.xml. But these starter dependencies lets you get up and running in no time.



These starter dependency templates are an important part of Spring Boot. So what exactly are they? These are built-in dependency descriptors that you include as a part of the pom.xml file for your Spring Boot application. A starter dependency descriptor will contain the granular dependencies and versions for all of the components that you need to build that particular kind of app. It'll pull in and manage all dependencies, including transitive dependencies.

Spring Boot Microservices: Getting Started

Spring Boot: Starter Dependencies



With these starter dependency descriptors, all you need to know is the kind of Spring application that you're looking to build. Let's say you want to build a Spring Boot web application, `spring-boot-starter-web` is the dependency that you'll specify. This will pull in all of the individual libraries that you need, including third party integrations with Apache Tomcat, Jetty, and Undertow.

If you want to include unit test and other kinds of test along with your Spring application, you will also include the `spring-boot-starter-test` dependencies. You can specify multiple starter dependency templates.

Let's say you're planning to build a UI-based application and you need a template engine that allows you to render views. You can use the `spring-boot-starter-thymeleaf` dependency. And these are just a few examples. As you work through the demos of this course, you will find that there are many starter dependencies out there. You don't have to worry about the granular libraries and versions of those libraries. You simply specify the starter template which simplifies your `pom.xml` file.

Spring Boot Microservices: Getting Started

Spring Boot: Simplify pom.xml



Because you don't need to explicitly specify the third party dependency within this file. In addition, you don't need to specify the version number for each explicit dependency. You only specify the version for Spring Boot. And then Spring Boot will make sure that the versions of the individual dependencies all work with one another, they interoperate.

Most of the dependency management complexity is part of the parent pom.xml that Spring Boot manages and generates. It's not your responsibility as a user unless you want some very granular configuration. What you create will be the User pom.xml, which is fairly simple and only specifies the starter dependencies.

Spring vs. Spring Boot – Auto configuration



Let's discuss some more examples of how Spring Boot makes things easy. In Spring, as a developer, you need to specify the dispatcher servlet, mappings and other configurations within your web app. With Spring Boot, all of these beans are created via auto-configuration.



Spring Boot Microservices: Getting Started

What exactly is auto-configuration? This is a way that Spring Boot users to automatically configure your Spring application based on the jars and other Java dependencies that are found on the classpath of the application.

Spring Boot Auto Configuration



Spring Boot basically looks at the classpath of your Java application and says, oh here are all of these jars. This Spring application needs these jars, make sure they're automatically available to the Spring application. The auto-configuration feature in Spring Boot relies on specific Spring annotations, which are variants of the `@Configuration` annotation. Each `@Configuration` annotation is an opinionated default specified by your app. Now, these `@Configuration` annotations are used conditionally, which means that if you have overridden the default specification for some property, your overridden version will be used, not the default. Thus, Spring Boot respects user-provided defaults. So if you don't want the default specification, you can override it. Otherwise, the default specification is there for you to use.



Spring Boot auto-configuration automatically detects and adds beans that you haven't explicitly manually configured. Within a Spring Boot app, you need to opt-in to this auto-configuration by adding `@EnableAutoConfiguration` or `@SpringBootApplication`, to the entry point of your app. When we write code in just a little bit, you'll see that we'll discuss these annotations in a lot more detail. Now, auto-

Spring Boot Microservices: Getting Started

configuration is non-invasive, which means you automatically get sensible defaults. But you can replace individual implementations by defining your own beans wherever you want to. You can disable auto-configured default classes in specific instances using either annotation attributes or properties that you specify in an XML file.



The auto-configured components and dependencies in your app are applied conditionally. They are `ConditionalOnClass` so check dependencies on specific classes before creating the auto-configured bean. Spring Boot does this automatically. They can also be conditional on a specific property. Check dependencies on properties before creating the auto-configured bean. This means that the bean will be auto-configured only if a specific property exists within your app.

Auto-configuration can also be `ConditionalOnMissingBean`. So create an auto-configured bean only if no user-specific bean is available. Spring Boot will correctly use the default only if no user implementation has been provided.

Spring vs. Spring Boot – Templates

<ul style="list-style-type: none">• Spring uses Thymeleaf template engine• Dependency and configuration for view-resolver must be done by developer	<ul style="list-style-type: none">• Spring Boot also uses Thymeleaf template engine• Developer merely needs to specify one dependency<ul style="list-style-type: none">• <code>spring-boot-starter-thymeleaf</code>
--	--

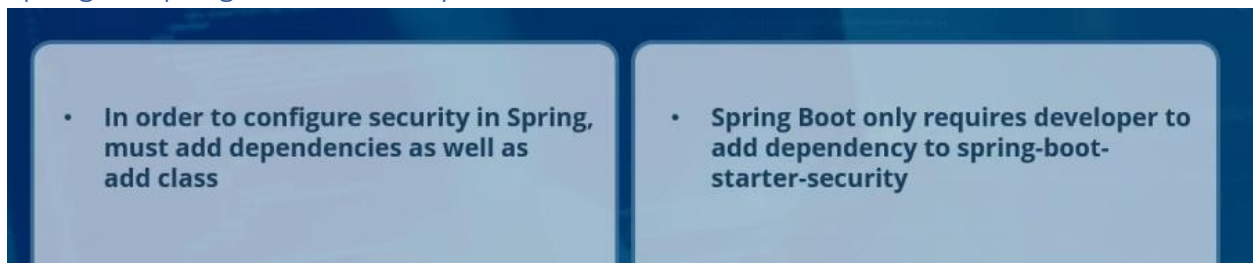
When we work with Spring Boot to build user interface based applications we'll extensively use the Thymeleaf template engine which was also available in Spring. So both Spring and Spring Boot use Thymeleaf. But with Spring, the dependency and configuration for the view-resolver must be done explicitly by the developer. But when you use the Thymeleaf template engine with Spring Boot, you only need to specify one starter dependency, `spring-boot-starter-thymeleaf`. And the Thymeleaf template engine will be available for you to use.

Spring Boot Microservices: Getting Started



So what exactly is this Thymeleaf Template Engine? Well, this is a server-side Java template engine that can process HTML, XML, CSS, JavaScript, and plain text. Thymeleaf has a large variety of control structures that you can use to transform your data before it's rendered to the user.

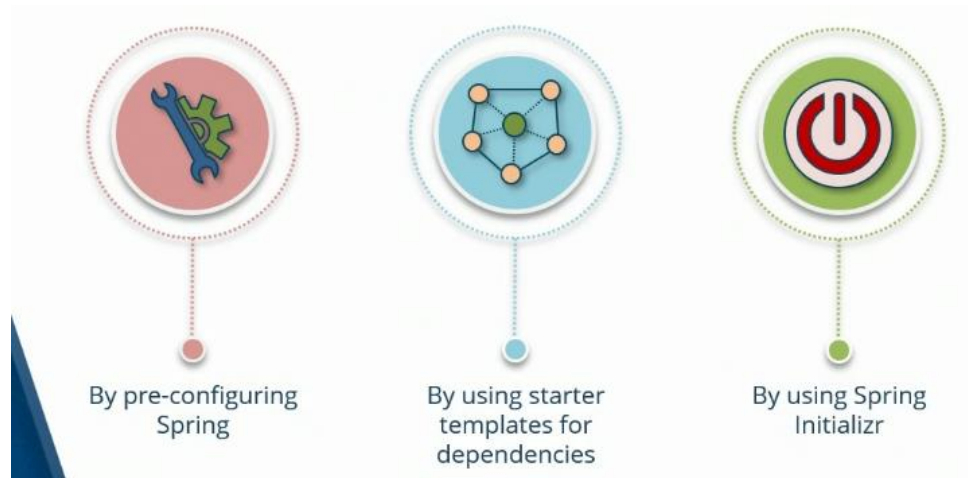
Spring vs. Spring Boot – Security



You'll also find that integrating with the Spring security module is much easier in Spring Boot than in Spring. To configure security in Spring, you must add dependencies as well as classes. But with Spring Boot, you only need to specify the starter template spring-boot-starter-security.

Spring Boot Microservices: Getting Started

Spring Initializr



Spring Boot makes working with Spring easier by pre-configuring the Spring modules that you need. By giving you starter templates that you can use for dependency management and version control. And finally, Spring Boot also simplifies Spring by allowing you to use the Spring Initializr.



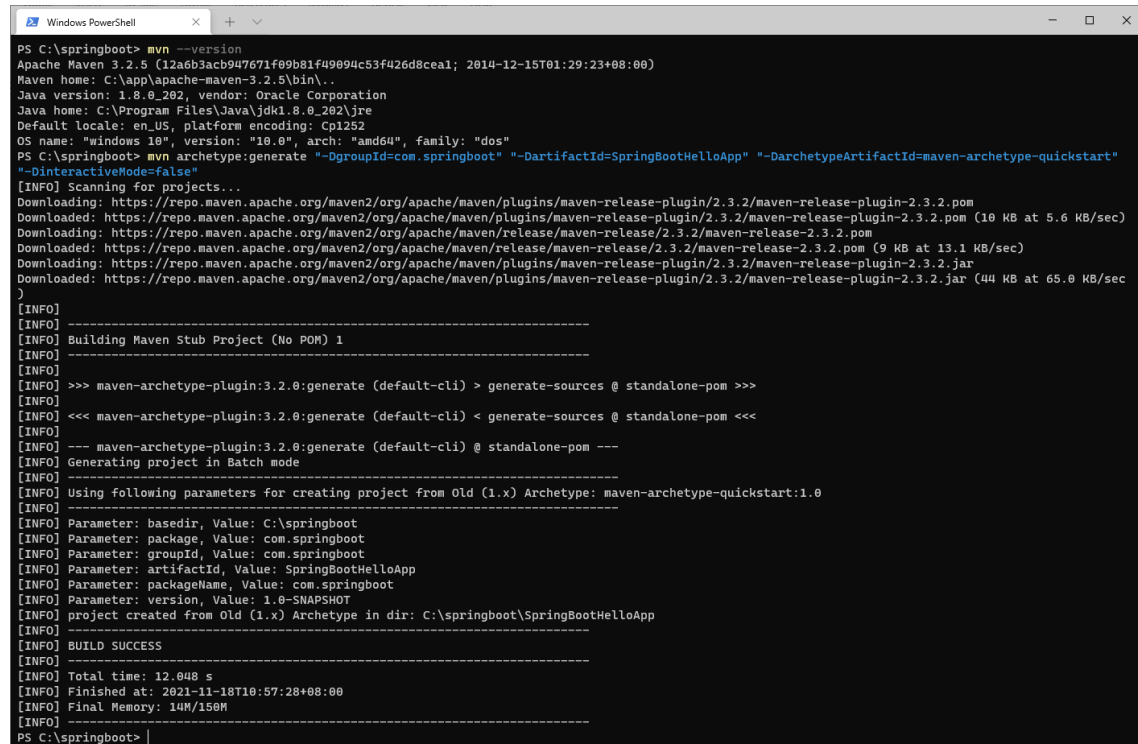
The Spring Initializr is basically just a web interface that you can use to set up your initial project structure for Spring Boot. Instead of manually setting up your Spring Boot project, you can use the Spring Initializr and download a zip file with your project. And all of the dependencies that you've specified set up for you automatically.

Spring Boot Microservices: Getting Started

Importing Maven Project

1. Generate Maven Project using Maven

We'll generate our first Maven project using the `mvn archetype:generate` command. A Maven archetype is a Maven project templating tool kit. An archetype is defined as any original pattern or model from which all other things of the same kind are made.



```
PS C:\springboot> mvn --version
Apache Maven 3.2.5 (12a6b3ac947671f09b81f49094c53f426d8ceal; 2014-12-15T01:29:23+08:00)
Maven home: C:\app\apache-maven-3.2.5\bin\..
Java version: 1.8.0_202, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_202\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "dos"
PS C:\springboot> mvn archetype:generate "-DgroupId=com.springboot" "-DartifactId=SpringBootHelloApp" "-DarchetypeArtifactId=maven-archetype-quickstart"
"-DinteractiveMode=false"
[INFO] Scanning for projects...
Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-release-plugin/2.3.2/maven-release-plugin-2.3.2.pom
Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-release-plugin/2.3.2/maven-release-plugin-2.3.2.pom (10 KB at 5.6 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/release/maven-release/2.3.2/maven-release-2.3.2.pom
Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/release/maven-release/2.3.2/maven-release-2.3.2.pom (9 KB at 13.1 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-release-plugin/2.3.2/maven-release-plugin-2.3.2.jar
Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-release-plugin/2.3.2/maven-release-plugin-2.3.2.jar (44 KB at 65.0 KB/sec)
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO]
[INFO] >>> maven-archetype-plugin:3.2.0:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:3.2.0:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:3.2.0:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO]
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO]
[INFO] Parameter: basedir, Value: C:\springboot
[INFO] Parameter: package, Value: com.springboot
[INFO] Parameter: groupId, Value: com.springboot
[INFO] Parameter: artifactId, Value: SpringBootHelloApp
[INFO] Parameter: packageName, Value: com.springboot
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\springboot\SpringBootHelloApp
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
[INFO] -----
[INFO] Total time: 12.048 s
[INFO] Finished at: 2021-11-18T10:57:28+08:00
[INFO] Final Memory: 14M/150M
[INFO] -----
PS C:\springboot>
```

Here we're using the Maven archetype Quick Start, as we have specified in the archetype artifact ID in order to generate a template for our starter project. When we generate our template we need to specify the group ID and the artifact ID for our project. The group ID uniquely identifies our project within a group of project and it's typically the reverse domain name and is also the package under which you will work in your Java code.

mvn archetype:generate

- DgroupId={project-packaging}
- DartifactId={project-name}
- DarchetypeArtifactId=**maven-archetype-quickstart**
- DinteractiveMode=**false**

So groupId is **com.springboot**. The Maven artifactId is the name of the jar file that you will build using your Java project without a version specification. So our artifactId here is **SpringBootHelloApp**. This is our first app after all.

Go ahead and hit Enter and a new Maven project will be generated for you with some starter code that you can work with. Once you see **BUILD SUCCESS**, you can run an *ls command* and

Spring Boot Microservices: Getting Started

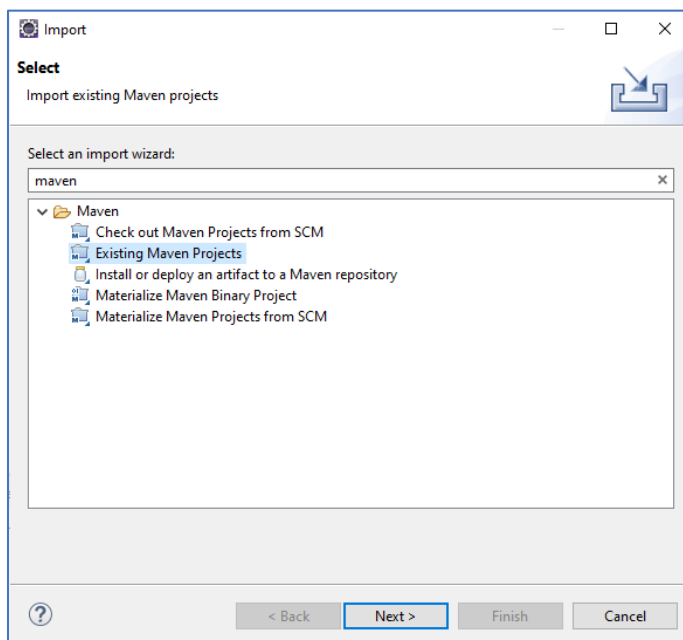
you'll see that a new sub folder has been created in the current working directory corresponding to your project.

```
PS C:\springboot> tree /A /F
Folder PATH listing for volume OSDisk
Volume serial number is E6DA-7971
C:.\
|--SpringBootHelloApp
    |--pom.xml
    |--src
        |--main
            |--java
                |--com
                    |--springboot
                        App.java
        |--test
            |--java
                |--com
                    |--springboot
                        AppTest.java
```

Let's take a look at this SpringBootHelloApp directory using our finder or explorer window. You can expand this folder and you'll see the basic structure of a Java project has been set up for you. You can see the **pom.xml** file there. This is the project object model file in Maven, this is where you'll specify the dependencies for your project.

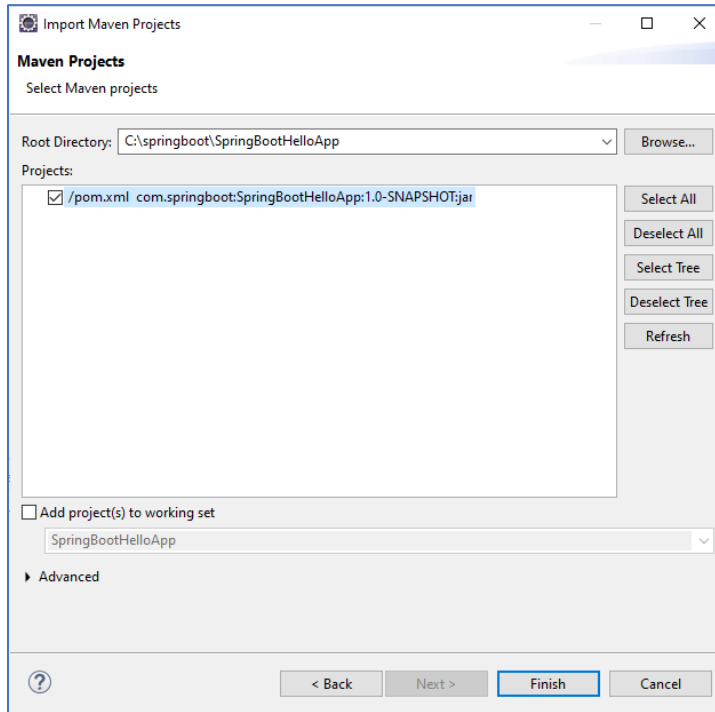
2. Import Maven Project inside Eclipse

The Java code for your source files and test files you will write within this source directory. You can expand this further and you will see that a stub for App.java has been created for you. This is a simple Hello World application. In this learning path, we'll be working with the Eclipse IDE. Switch over and start your Eclipse IDE. And let's use the File Import command to import our Maven project into Eclipse.

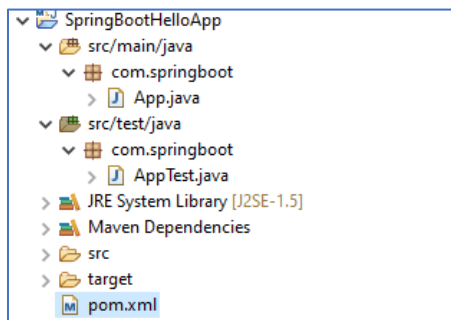


Spring Boot Microservices: Getting Started

You'll see a wizard which looks like this and you will see an option for Maven, expand this option and select Existing Maven Projects. Click on the Next button. This will take you to a page where you specify the folder for your existing Maven project. Click on the Browse button. This will bring up your Explorer window, and browse to the subdirectory where your project has been created. The directory that you select here is the directory that contains your pom.xml file SpringBootHelloApp.



Once you've specified this, click on the Finish button, and your existing Maven project will be imported into your Eclipse IDE. Project will appear as a node within your project explorer pane off to the left, you can expand this project and see the structure of the project within which you're going to write code. The pom.xml is available there as well.



Spring Boot Microservices: Getting Started

Building a Spring Boot Application

3. Update Dependency Library in POM.xml

Let's get started with our very first Spring Boot Application. Now the Spring framework basically offers everything that you need to build complex Java applications. And in recent years, the Spring framework has gotten really complex. It offers application frameworks for web applications, accessing databases, security, and so on. Spring Boot, is what makes it easy for you to get up and running with your Spring projects. So you will find that when you use Spring Boot for your Spring projects, it's almost like magic, you're up and running in no time at all. And you will see this magic begin in the pom.xml file.

When you work with Spring Boot, it's best to use a dependency management system, such as Apache Maven or Gradle. Other systems also work but these are the most common ones, and they are well tested, and will work for you almost out of the box.

Here's what the default form looks like generated by my Maven starter project. I'm going to update this to include Spring Boot dependencies.

pom.xml



```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.3.1.RELEASE</version>
9     <relativePath></relativePath>
10  </parent>
11
12  <groupId>com.springboot</groupId>
13  <artifactId>SpringBootHelloApp</artifactId>
14  <packaging>jar</packaging>
15  <version>1.0-SNAPSHOT</version>
16  <name>SpringBootHelloApp</name>
17  <url>http://maven.apache.org</url>
18
19  <dependencies>
20    <dependency>
21      <groupId>junit</groupId>
22      <artifactId>junit</artifactId>
23      <!-- <version>3.8.1</version> -->
24      <scope>test</scope>
25    </dependency>
26
27    <dependency>
28      <groupId>org.springframework.boot</groupId>
29      <artifactId>spring-boot-starter</artifactId>
30    </dependency>
31  </dependencies>
32 </project>
```

The first thing that you ought to observe here is that, I have a parent tag on lines 5 through 10, which specifies the **spring-boot-starter-parent**. And there is a version number **2.3.1.RELEASE** that is associated with this spring-boot-starter.

The whole idea of including this starter-parent dependency is that it provides sensible defaults for all of our Spring Boot dependencies in any configuration that we specify. Specifying this

Spring Boot Microservices: Getting Started

within the parent tag indicates to Maven that we want to accept all of the dependencies available by default in this starter-parent.

When you use the **spring-boot-starter-parent**, you only need to specify the version number for your parent dependency. It happens to be 2.3.1, the latest version at the time of this recording. You don't need to specify specific versions for all of your other Spring dependencies. All of the other Spring modules that you will work with, that you will depend on will simply get the right version from this starter-parent. So you don't have to worry about the versions of individual modules, which is a great load off your mind.

Once you've inherited from the spring-boot-starter-parent, for sensible defaults, another key feature of using Spring Boot for your application is the use of starter dependency descriptors. And the *dependency* descriptor that we've defined here is the **spring-boot-starter** on line 28.

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

Starter dependency descriptors are what makes Spring Boot so easy to work with. Instead of manually specifying the dependencies for your application, you simply specify the starter descriptor for the kind of application that you want to build. For a very basic Spring application where you depend on only the core Spring modules, this starter Spring Boot starter is sufficient.

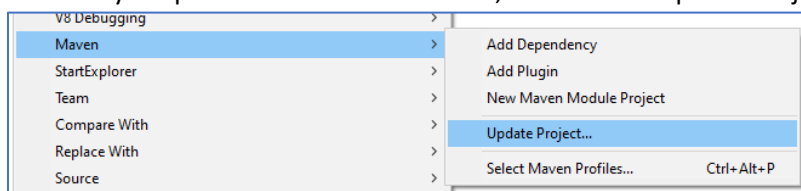
You will use different starters for different kinds of applications. For example, there is a web starter for web applications using Spring MVC, there is a JPA starter. If you want to use an object relational mapping model to work with your relational database. When you use these starters, you're working with production-ready, tested and supported dependency configurations. You don't have to worry about version management of individual Spring modules, everything will just work.

Another thing to note here, before we move on from this pom.xml file, is on line 14.

```
<packaging>jar</packaging>
```

Notice that our application is built as a jar. That is, it is a self contained Java application. You don't have to explicitly deploy the application somewhere for it to run, Spring will just work.

Please do update project in your eclipse after modifying dependencies **pom.xml**, by do "right click" on your pom.xml and select Maven, and click on Update Project.

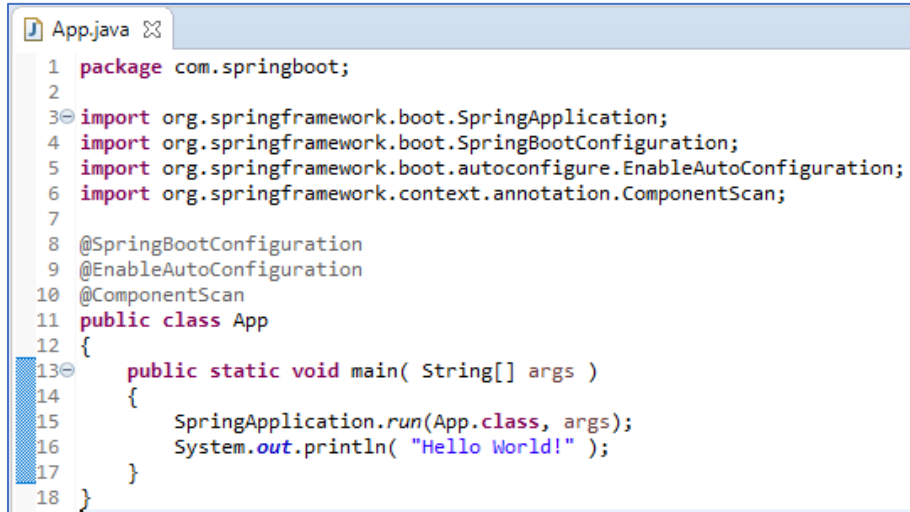


Spring Boot Microservices: Getting Started

4. Update Main Method in App.java Class

Now let's go ahead and take a look at our App.java code, and write our very first Spring Boot application here. Update the code that has been generated by default here and set up my Spring code as follows.

App.java



```
1 package com.springboot;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.context.annotation.ComponentScan;
6
7 @SpringBootApplication
8 @EnableAutoConfiguration
9 @ComponentScan
10 public class App
11 {
12     public static void main( String[] args )
13     {
14         SpringApplication.run(App.class, args);
15         System.out.println( "Hello World!" );
16     }
17 }
18 }
```

Now the first thing that ought to jump out at you is the three annotations that we've specified on this app class. Each of the annotations that we've specified here on lines 8, 9, and 10 has its own role to play.

@SpringBootApplication
@EnableAutoConfiguration
@ComponentScan

And it's important that you understand what each annotation does because these form the core of the Spring Boot framework.

- The first of these annotations here is the **@SpringBootApplication**. This is a class level annotation that is part of the Spring Boot framework, and it indicates that a class provides application configuration.
Now in Spring Boot, you will typically use Java annotation-based configuration specification rather than XML files. As a result, this **@SpringBootApplication** annotation is the primary source for configuration within your applications. Generally, you will apply this annotation to the class which contains the **main method**, the **entry point** for your Spring Boot application.
The use of this annotation in the class that is your entry point allows configuration to be automatically located. So that's where this is typically used.
- The next annotation that we've applied here is the **@EnableAutoConfiguration**. Now, Spring Boot frees you up from dependency management. Once you specify the starter descriptors within your pom.xml, all of the dependencies that are part of those starter descriptors are

Spring Boot Microservices: Getting Started

automatically available in your Java classpath. Now when you apply this **@EnableAutoConfiguration** annotation to your Spring Boot app, this auto-configures the beans present in the Java classpath.

Any dependency available in the classpath of your application set up using your starter descriptors will be injectable automatically as a bean within your app. And you can reference those libraries in a very straightforward manner without jumping through hoops.

Auto-configuration does a lot of magic behind the scenes by guessing the beans that you require. For example, if you have the Tomcat embedded jar in your class path, then you will need a Tomcat embedded servlet container factory bean to configure the Tomcat server. This will be injected automatically. This is done without you having to specify any specific XML-based or Java-based configuration.

- And finally, we come to the third annotation here, the **@ComponentScan**. This basically tells Spring Boot that you need to scan the current package, which in our case is *com.springboot*, and all of the sub packages within this current package for injectable beans.

Spring Boot will take care of performing the scan and injecting all of the beans that are our dependencies.

As you can see, all of these three annotations that we've specified do a lot of work for us behind the scenes. They are important specifications and no Spring Boot project is complete without these. The actual app itself is very straightforward. We have the entry point, the public static void main method,

```
public static void main( String[] args )
```

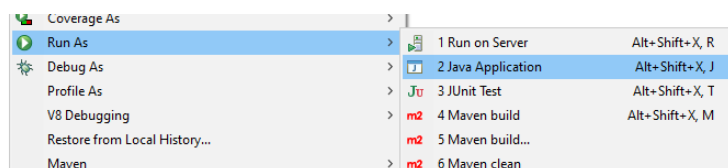
we simply use `SpringApplication.run` to run this class.

```
SpringApplication.run(App.class, args);  
System.out.println( "Hello World!" );
```

And the only thing that this application does is print out to the console window, Hello World! it's a very simple app.

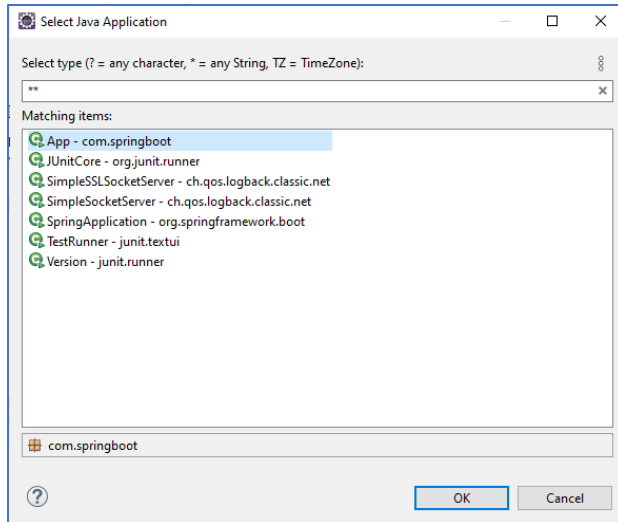
5. Run to test the Application

Let's now run this code, you can right click on the `SpringBootHelloApp` project, go to Run As and Java Application.

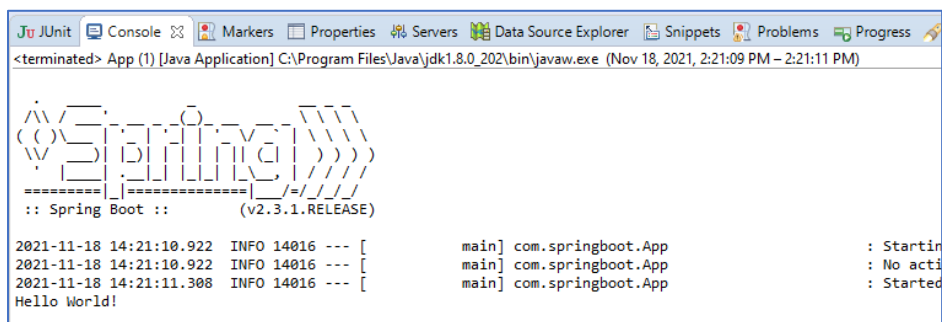


Remember, Spring Boot applications are just jar files.

Spring Boot Microservices: Getting Started



You'll find that Spring starts up, the app that we want to run is *com.springboot*, select that and hit OK.



Console messages tell you that Spring is starting up, this is version 2.3.1.RELEASE. And at the very bottom, we have the Hello World! from our system out println statement.

Now this application was very straightforward so there was nothing much to configure. But everything that we needed for this application was set up for us by default, using just the starter dependencies that we had specified.

Spring Boot Microservices: Getting Started

6. Implement Simpler Annotation in entry method to Run Spring Boot Application

Now when you're working in Spring Boot, it's actually more common to use a single annotation which encompasses all three of the annotations that we saw earlier, the

@SpringBootApplication annotation that you see applied to this App class. The

@SpringBootApplication basically does the work of all three annotations that we saw earlier.

```
App.java
1 package com.springboot;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class App
8 {
9     public static void main( String[] args )
10    {
11        SpringApplication.run(App.class, args);
12        System.out.println( "Hello World!" );
13    }
14 }
```

And if you run this code, you'll see Hello World! printed out to screen.

```
JUnit Console Markers Properties Servers Data Source Explorer Snippets Problems Progress Search
<terminated> App (1) [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 18, 2021, 2:30:16 PM - 2:30:18 PM)

:: Spring Boot :: (v2.3.1.RELEASE)

2021-11-18 14:30:17.779 INFO 25508 --- [main] com.springboot.App : Starting App c
2021-11-18 14:30:17.782 INFO 25508 --- [main] com.springboot.App : No active prof
2021-11-18 14:30:18.158 INFO 25508 --- [main] com.springboot.App : Started App in
Hello World!
```

Everything is exactly the same. How do we know that this **@SpringBootApplication** annotation is a combination of the three annotations that we saw earlier? I'm going to right click on this, go to Open Declaration and this will open up the class file for this annotation.

```
SpringBootApplication.class
48 * @author Andy Wilkinson
49 * @since 1.2.0
50 */
51 @Target(ElementType.TYPE)
52 @Retention(RetentionPolicy.RUNTIME)
53 @Documented
54 @Inherited
55 @SpringBootApplication
56 @EnableAutoConfiguration
57 @ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
58     @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
59 public @interface SpringBootApplication {
60 }
```

Observe on lines 55 through 58, the three original annotations that we had applied to our Spring Boot app is present here, **SpringBootApplication**, **EnableAutoConfiguration**, and **@ComponentScan**. So rather than explicitly specifying these three annotations separately, it's common practice to simply annotate your main method class using **@SpringBootApplication**.

Spring Boot Microservices: Getting Started

Running on the Embedded Tomcat Server

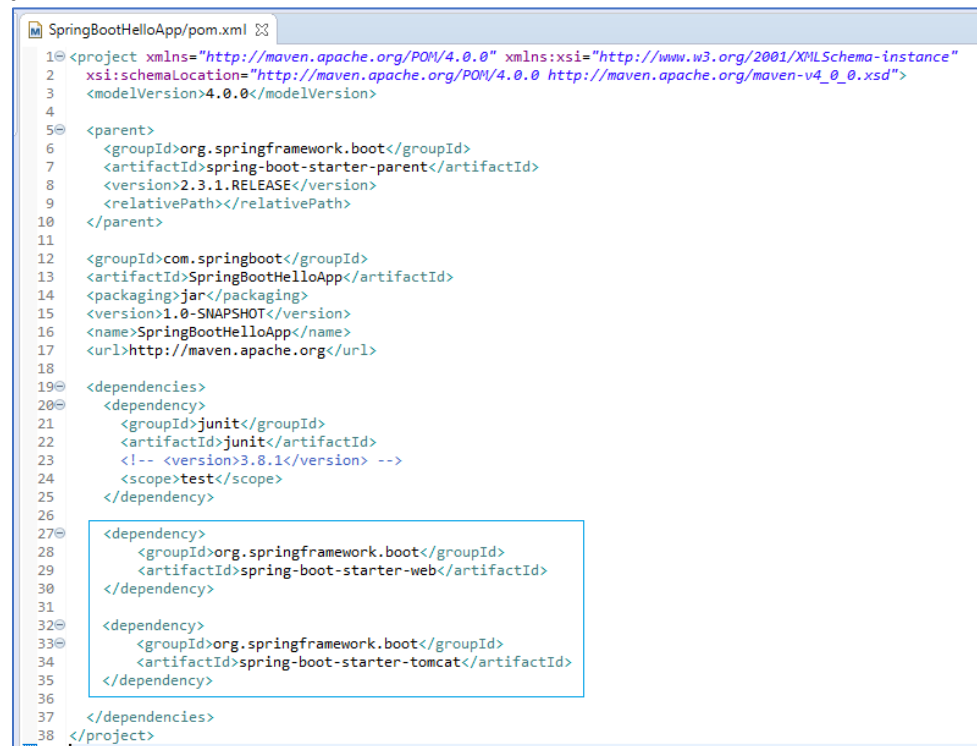
In this demo, we'll take a look at how we can build a very simple Spring MVC application using Spring Boot.

1. Update Dependency in Pom.xml

The dependencies that we specify in our pom.xml is what will be different.

Let's take a look here. Within pom.xml, we still have the parent tag.

pom.xml



```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.3.1.RELEASE</version>
9     <relativePath></relativePath>
10  </parent>
11
12  <groupId>com.springboot</groupId>
13  <artifactId>SpringBootHelloApp</artifactId>
14  <packaging>jar</packaging>
15  <version>1.0-SNAPSHOT</version>
16  <name>SpringBootHelloApp</name>
17  <url>http://maven.apache.org</url>
18
19  <dependencies>
20    <dependency>
21      <groupId>junit</groupId>
22      <artifactId>junit</artifactId>
23      <!-- <version>3.8.1</version> -->
24      <scope>test</scope>
25    </dependency>
26
27    <dependency>
28      <groupId>org.springframework.boot</groupId>
29      <artifactId>spring-boot-starter-web</artifactId>
30    </dependency>
31
32    <dependency>
33      <groupId>org.springframework.boot</groupId>
34      <artifactId>spring-boot-starter-tomcat</artifactId>
35    </dependency>
36
37  </dependencies>
38 </project>
```

We depend on spring-boot-starter-parent to give us sensible defaults for our app. This is the only place where we need to specify the version of Spring Boot that we use, 2.3.1.RELEASE. In all other dependency specification, we can eliminate the version.

```
<version>1.0-SNAPSHOT</version>
```

Spring Boot will provide us the right version numbers for all of our modules. Because this is going to be a web application, the starter dependency descriptor that we specify is different.

Notice on line 28, we have a dependency on spring-boot-starter-web.

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
```

This is what you'll use for all your web applications. The starter web dependency automatically allows us to use *Spring MVC* to develop our web application, the Tomcat server to host our application, as well as Jackson to work with JSON. We specified another starter descriptor here on line 33, spring-boot-starter-tomcat.

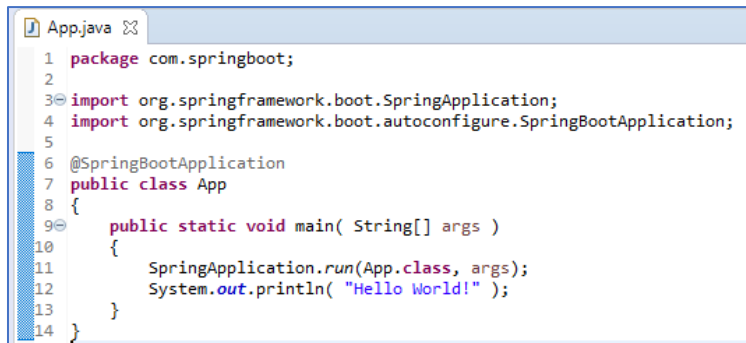
Spring Boot Microservices: Getting Started

This is the dependency that you'll specify if you want to work with the Tomcat server, but you will see that *this is not really needed*. I've explicitly specified it here so that you know what is the starter descriptor for Tomcat servers. It's not needed because spring-boot-starter-web encompasses starter-tomcat.

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
```

2. Entry Class Implementing SpringBootApplication Annotation

Let's head over and take a look at the main entry point of our application which is the App.java file. Make sure you have the @SpringBootApplication annotation applied to this class so that you can make full use of all of the magic that it offers,

A screenshot of a code editor showing the App.java file. The code is as follows:

```
1 package com.springboot;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class App
8 {
9     public static void main( String[] args )
10    {
11        SpringApplication.run(App.class, args);
12        System.out.println( "Hello World!" );
13    }
14 }
```

automatically injectable beans, auto-configuration and so on. All we do within this class is call SpringApplication.run on App.class. I've also printed out Hello World! out to screen within our console window.

3. Prepare Controller Class

When we use Spring MVC to build a web application, it's built using the Model-View-Controller paradigm. The Model-View-Controller is a standard design pattern used to build user interface based applications. The model is the part of your application that deals with databases and the underlying data that is represented.

The view is what is rendered to the user, and the controller contains the business logic of your app.

Spring Boot Microservices: Getting Started

HelloController.java

```
1 package com.springboot.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestMethod;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class HelloController {
10
11     @GetMapping("/")
12     public String index() {
13         return "This is the main page";
14     }
15
16     @RequestMapping(value = "/welcome", method = RequestMethod.GET)
17     public String welcome() {
18         return "Welcome to Spring Boot!";
19     }
20
21     @GetMapping("/hello")
22     public String hello() {
23         return "Hello Spring Boot!";
24     }
25 }
```

I've defined a simple HelloController here in my Spring MVC application which is a simple rest controller. Notice the **@RestController** annotation on my HelloController.

A rest controller essentially is what you'll use to build up your Rest APIs, Create, Read, Update and Delete actions. When you tag your class using **@RestController**, Spring MVC essentially knows that whatever response you send from the controller methods that have been mapped are essentially web responses that need to be rendered to the user as such.

You'll understand what I mean in more detail further down this learning path, where we see the difference between *Controllers* and *RestControllers*.

Now all of the methods that we have specified within this HelloController are essentially handler mappings for incoming requests. Every method handles a specific request and we use the **@RequestMapping** annotation to specify the path of the incoming request.

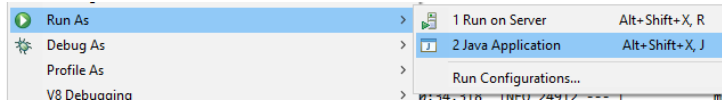
This is a very simple REST API which only responds to GET methods, and we have three handler mappings configured.

Whenever a request is made to any of the parts that we have mapped here within this controller, the right response will be displayed to the user.

4. Run and Test the Stand alone Web Application

Now in order to run this web application, you don't have to build a WAR file or anything like that. Simply run this code as a Java application and a Tomcat server will be spun up automatically and your application will be hosted on that Tomcat server.

Spring Boot Microservices: Getting Started

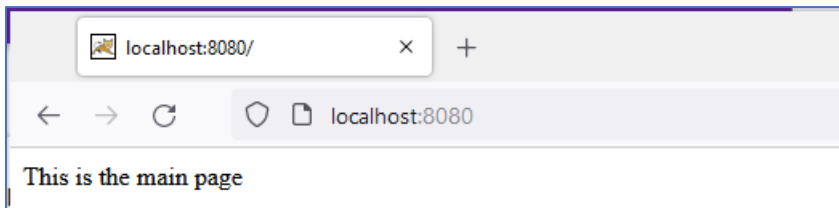


You can see Hello World! printed here at the console tab.



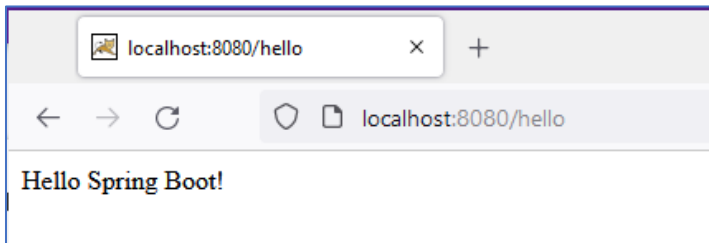
And if you take a look at the other console messages, you can see that our Tomcat server running at localhost 8080/ has been spun up automatically. This is the embedded Tomcat server available as a part of Spring Boot. This embedded Tomcat server automatically spins up the Catalina execution engine for its servlets. And that's it, you have your web application hosted within Tomcat.

Let's go to <http://localhost:8080> and there you'll see the main page.



Using the browser, we made a GET request to the path /, the index handler method was invoked. And this is the main page was returned to the user and rendered as a web response.

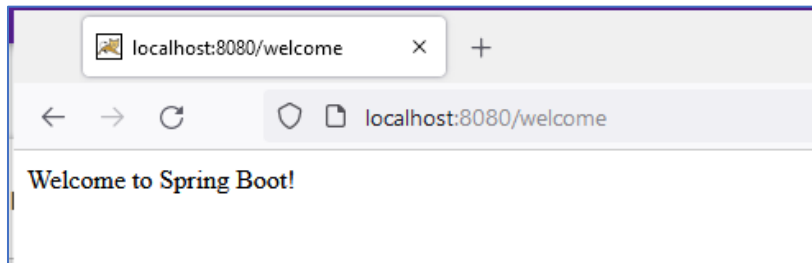
Let's take a look at [localhost/hello](http://localhost:8080/hello).



This time around, the handler method corresponding to the path /hello was invoked and Hello Spring Boot! is printed out to our browser window.

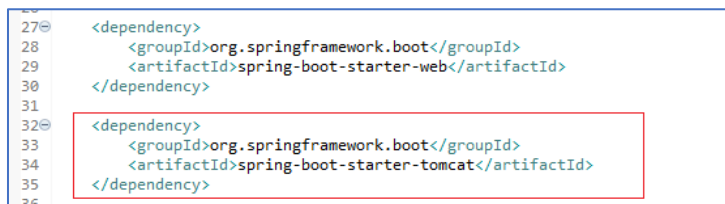
Spring Boot Microservices: Getting Started

Let's try one last time to the path /welcome

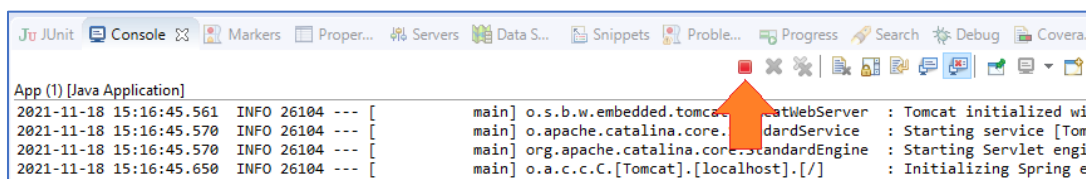


This invokes the welcome method corresponding to the path /welcome and Welcome to Spring Boot! is rendered to screen.

Now, let's make one slight change. Remember in the pom.xml, I had said that the starter-tomcat dependency is not really needed, because Tomcat is available by default as a part of the starter-web dependency descriptor. I'll get rid of Tomcat, we'll only keep the starter-web dependency descriptor, and you will find that things work just fine.



Now before you run the server again, make sure you use this big red button to stop the previously running server before you restart Tomcat.



If you scroll down below, you'll see that the Tomcat server is started up and our web application is hosted within Tomcat. The Tomcat server is running on port 8080. Now if you head over to your browser window, you'll see that your application works as it did before.

You don't explicitly need the Tomcat dependency when you're working with the starter-web dependency descriptor. /hello works, and you will see that /welcome also works. With the starter web dependency, we're using the embedded Tomcat server that it makes available.

Spring Boot Microservices: Getting Started

Running on The Jetty Server

In the previous demo, we saw how we could host our web application using the default embedded Tomcat server available as a part of the Spring Boot starter web dependency. But what if within your application you want to use one of the other embedded servers that are available?

There are three embedded servers available to host your applications, when you use the Spring Boot starter web dependency descriptor.

There are the Apache Tomcat server which we saw in the previous demo.

You also have the Jetty server and the Undertow server available to you by default.

But how do you use one of these other servers rather than Tomcat, and that's exactly what we'll see here in this demo.

1. Update Dependency in Pom.xml

Here is my **pom.xml**. Notice that I've included the Spring Boot starter parent.

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.3.1.RELEASE</version>
9     <relativePath></relativePath>
10  </parent>
11
12  <groupId>com.springboot</groupId>
13  <artifactId>SpringBootHelloApp</artifactId>
14  <packaging>jar</packaging>
15  <version>1.0-SNAPSHOT</version>
16  <name>SpringBootHelloApp</name>
17  <url>http://maven.apache.org</url>
18
19  <dependencies>
20    <dependency>
21      <groupId>junit</groupId>
22      <artifactId>junit</artifactId>
23      <!-- <version>3.8.1</version> -->
24      <scope>test</scope>
25    </dependency>
26
27    <dependency>
28      <groupId>org.springframework.boot</groupId>
29      <artifactId>spring-boot-starter-web</artifactId>
30
31      <exclusions>
32        <exclusion>
33          <groupId>org.springframework.boot</groupId>
34          <artifactId>spring-boot-starter-tomcat</artifactId>
35        </exclusion>
36      </exclusions>
37    </dependency>
38
39    <dependency>
40      <groupId>org.springframework.boot</groupId>
41      <artifactId>spring-boot-starter-jetty</artifactId>
42    </dependency>
43
44  </dependencies>
```

This is going to be the default for all of the Spring Boot applications that we'll build. If you scroll down below, you'll see something interesting. On line 28, I still have the Spring Boot starter web dependency descriptor, I want to use spring MVC and I want to use their embedded servers.

Spring Boot Microservices: Getting Started

But I'm not interested in the default embedded server Tomcat, so I explicitly **exclude** on line 31 to 36, the Spring Boot starter Tomcat dependency.

The use of the `exclusions` tag tells Maven that I want every dependency in Spring Boot starter web except for those dependencies specified in `spring-boot-starter-tomcat`. So I do not want Tomcat, instead on lines 39 to 42, I specify the dependency that I am interested in, **spring-boot-starter-jetty**.

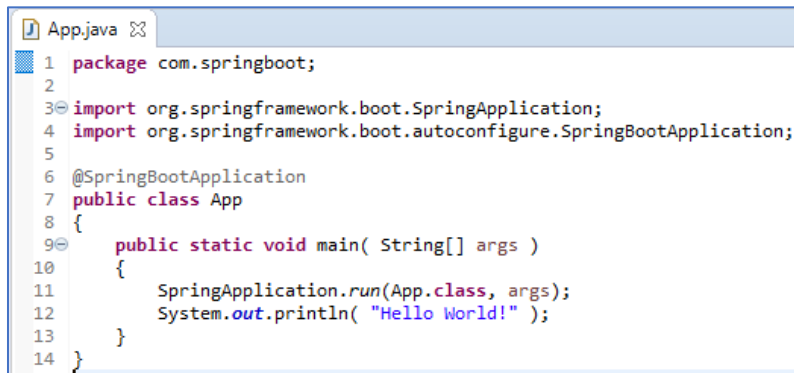
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Maven knows how to correctly interpret this. Include everything in `spring-boot-starter-web`, *except* for the dependencies in `spring-boot-starter-tomcat`. Instead of that include `spring-boot-starter-jetty`, which means we'll use the embedded Jetty server available in the starter web, rather than the embedded Tomcat server. And really this is the only change that we'll make to our code.

2. Implementation of Main Entry point of the Application

If you take a look at `App.java`, it's the same. We print out Hello World!, we use `SpringApplication.run` for `App.class`.

App.java

A screenshot of a code editor showing the `App.java` file. The code is as follows:

```
1 package com.springboot;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class App
8 {
9     public static void main( String[] args )
10    {
11        SpringApplication.run(App.class, args);
12        System.out.println( "Hello World!" );
13    }
14 }
```

Notice the `@SpringBootApplication` annotation on the class itself.

Spring Boot Microservices: Getting Started

3. Implementation of Controller Class

HelloController.java

```
1 package com.springboot.controller;  
2  
3 import org.springframework.web.bind.annotation.GetMapping;  
4 import org.springframework.web.bind.annotation.RequestMapping;  
5 import org.springframework.web.bind.annotation.RequestMethod;  
6 import org.springframework.web.bind.annotation.RestController;  
7  
8 @RestController  
9 public class HelloController {  
10  
11     @GetMapping("/")  
12     public String index() {  
13         return "This is the main page";  
14     }  
15  
16     @RequestMapping(value = "/welcome", method = RequestMethod.GET)  
17     public String welcome() {  
18         return "Welcome to Spring Boot!";  
19     }  
20  
21     @GetMapping("/hello")  
22     public String hello() {  
23         return "Hello Spring Boot!";  
24     }  
25 }
```

Nothing changes in our controller specification as well where we specify the handler path mappings. We have three handler mappings configured here, all methods respond to the HTTP request GET. We have the / which displays the main page, the /welcome, which says Welcome to Spring Boot! and the /hello which says Hello Spring Boot!.

Spring Boot Microservices: Getting Started

4. Run and Test the Application

Go ahead and run this code as a Java application. You can see Hello World! printed out to screen, everything seems to have worked fine. If you look at the console messages, that's where things are different.



```
App (1) [Java Application]
:: Spring Boot :: (v2.3.1.RELEASE)

2021-11-18 15:32:15.390 INFO 22280 --- [main] com.springboot.App : Starting App on US-5CD01445Q9 with PID 22280 (C:\springboot\SpringBoo
2021-11-18 15:32:15.394 INFO 22280 --- [main] com.springboot.App : No active profile set, falling back to default profiles: default
2021-11-18 15:32:17.418 INFO 22280 --- [main] o.s.b.w.e.j.JettyServletWebServerFactory : Server initialized with port: 8080
2021-11-18 15:32:17.422 INFO 22280 --- [main] org.eclipse.jetty.server.Server : jetty-9.4.29.v20200521; built: 2020-05-21T17:20:40.598Z; git: 77c232a
2021-11-18 15:32:17.452 INFO 22280 --- [main] o.e.j.s.h.ContextHandler.application : Initializing Spring embedded WebApplicationContext
2021-11-18 15:32:17.452 INFO 22280 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2011 ms
2021-11-18 15:32:17.624 INFO 22280 --- [main] org.eclipse.jetty.server.session : DefaultSessionIdManager workerName=node0
2021-11-18 15:32:17.624 INFO 22280 --- [main] org.eclipse.jetty.server.session : No SessionScavenger set, using defaults
2021-11-18 15:32:17.625 INFO 22280 --- [main] org.eclipse.jetty.server.session : node0 Scavenging every 60000ms
2021-11-18 15:32:17.635 INFO 22280 --- [main] o.e.jetty.server.handler.ContextHandler : Started o.s.b.w.e.j.JettyEmbeddedWebAppContext@344561e0{application,/
2021-11-18 15:32:17.636 INFO 22280 --- [main] org.eclipse.jetty.server.Server : Started @3289ms
2021-11-18 15:32:17.781 INFO 22280 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-11-18 15:32:17.922 INFO 22280 --- [main] o.e.j.s.h.ContextHandler.application : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-11-18 15:32:17.926 INFO 22280 --- [main] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-11-18 15:32:17.956 INFO 22280 --- [main] o.s.web.servlet.DispatcherServlet : Completed initialization in 4 ms
2021-11-18 15:32:17.956 INFO 22280 --- [main] o.e.jetty.server.AbstractConnector : Started ServerConnector@117e0fe5[HTTP/1.1, (http/1.1)]{0.0.0.0:8080}
2021-11-18 15:32:17.957 INFO 22280 --- [main] o.s.b.web.embedded.jetty.JettyWebServer : Jetty started on port(s) 8080 (http/1.1) with context path '/'
2021-11-18 15:32:17.965 INFO 22280 --- [main] com.springboot.App : Started App in 3.134 seconds (JVM running for 3.617)

Hello World!
```

Notice that the server that has started up is the Jetty web server. Jetty started on ports 8080 with context path/. We've seamlessly almost magically changed the web server on which our application is hosted by simply specifying a different starter dependency. You can make sure that this server runs as it did before, go to localhost:8080, you can see the main page displayed.

If you go to /hello, you'll see the Hello page displayed. And of course /welcome will give you the welcome page. And in exactly the same way if you want to use the embedded Undertow server, simply specify the spring-boot-starter-undertow dependency descriptor and exclude the embedded default Tomcat.

Spring Boot Microservices: Getting Started

Generating and Deploying and External WAR File

So far we've been deploying our Spring Boot web applications on the embedded servers that Spring Boot provides us. But what if we wanted to build a WAR file that is a web archive and deploy our web application on an external server? That's possible as well. You simply need to make a few changes to your pom.xml file and how your application is set up.

1. Update Dependency in Pom.xml

Let's take a look at pom.xml first.

pom.xml

The first thing we need to do is specify that we want to build a **WAR** file for our application. A WAR file is simply a web application archive file, it contains embedded within it all JAR files, as well as all of our HTML and other web resources.

A JAR file is a standalone Java application. A WAR file needs to be deployed and served on a web server. In order to tell Maven that we want to build a WAR file, you simply need to change the packaging.

On line 14, notice that we have specified WAR as the packaging rather than JAR.

Spring Boot Microservices: Getting Started

```
<packaging>war</packaging>
```

We're using Spring Boot to build our application even though it's now packaged as a WAR file. So we will continue to use the Spring Boot starter web dependency descriptor.

This is on line 28.

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

You'll now need some way to tell Spring Boot to not use the servlet container of the embedded Tomcat server. You want to indicate that an external Tomcat server will be provided for your application. And the way you do this is by explicitly specifying the Spring Boot starter Tomcat dependency with the scope provided.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <scope>provided</scope>
</dependency>
```

This effectively tells our application that the embedded servlet container dependency will be provided. We don't need to use the embedded Tomcat server, we'll be using an external Tomcat server.

Now if you want to ensure that your WAR files are also executable, you need to include the spring-boot-maven-plugin as you can see on line 41.

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

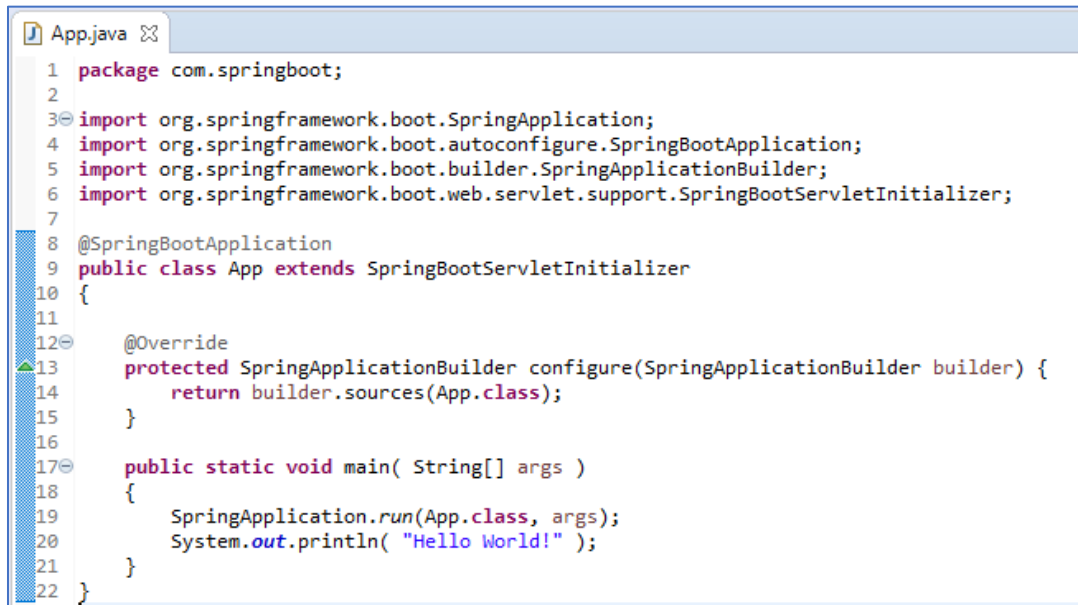
This plugin in Maven lets you package executable JAR or WAR archives to run an application in base.

2. Implementation of Main Entry Point of the Application

In order to create a deployable WAR file, you need to make a few changes to your application's code as well. Take a look at the main entry point, the App class, here notice that it extends the **SpringBootServletInitializer**.

Spring Boot Microservices: Getting Started

App.java



```
1 package com.springboot;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.boot.builder.SpringApplicationBuilder;
6 import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
7
8 @SpringBootApplication
9 public class App extends SpringBootServletInitializer
10 {
11
12     @Override
13     protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
14         return builder.sources(App.class);
15     }
16
17     public static void main( String[] args )
18     {
19         SpringApplication.run(App.class, args);
20         System.out.println( "Hello World!" );
21     }
22 }
```

Once we extend this base class, we need to override the `configure` method of this base class implementation, which we do on line 12.

```
@Override
protected SpringApplicationBuilder configure(SpringApplicationBuilder builder)
```

Setup is what allows us to run our spring application from a traditional WAR archive which has been deployed on an external web container. This class basically binds the *servlets filters*, the *servlet context* initializer beans from application context of our web app.

```
return builder.sources(App.class);
```

Within the overridden `configure` method, we simply register our current `App` class as a configuration class for our application. The `main` method, the entry point of our application, does not change.

We print out *Hello World!*. We also called `SpringApplication.run` for the current class.

Spring Boot Microservices: Getting Started

3. Implementation of Controller Class

There are no other code changes that we need to make within our web application.

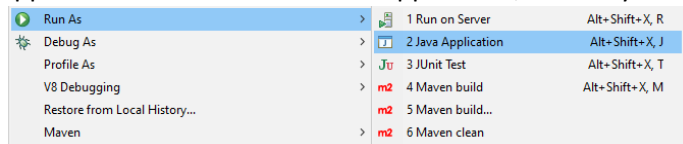
Here is our HelloController. It's a RestController. And we have three methods corresponding to three handler mappings to the /, /welcome and the /hello paths.

HelloController.java

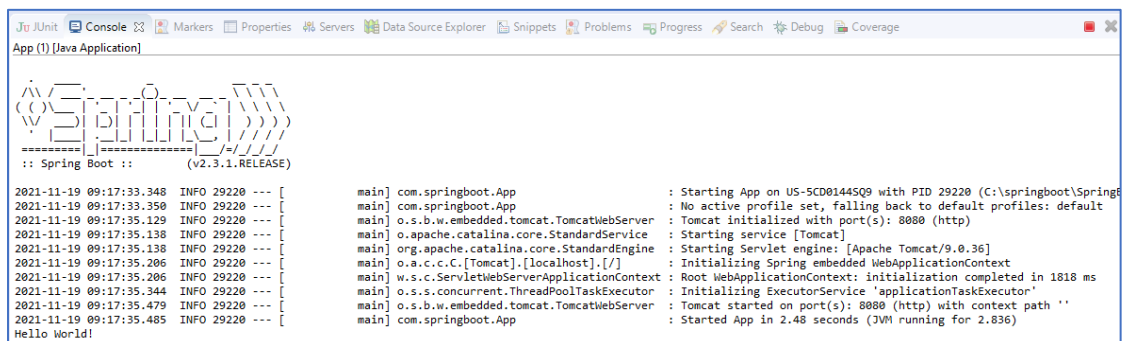
```
1 package com.springboot.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestMethod;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class HelloController {
10
11     @GetMapping("/")
12     public String index() {
13         return "This is the main page (using a wer deployment)";
14     }
15
16     @RequestMapping(value = "/welcome", method = RequestMethod.GET)
17     public String welcome() {
18         return "Welcome to Spring Boot! (using a wer deployment)";
19     }
20
21     @GetMapping("/hello")
22     public String hello() {
23         return "Hello Spring Boot! (using a wer deployment)";
24     }
25 }
```

4. Run to test the WAR application

Now even though we specified packaging as WAR, it's possible to run this code as a Java application as well. Run As Java Application, choose your current app.

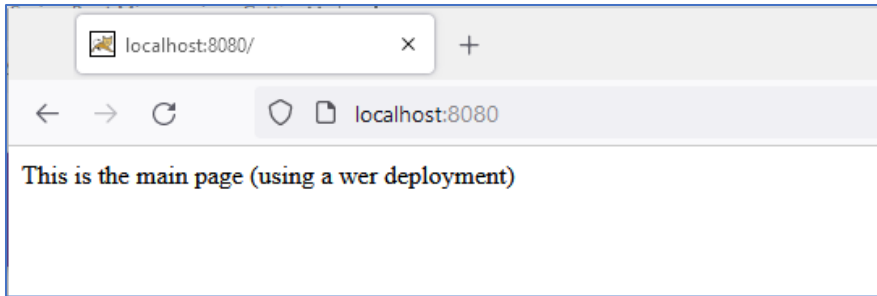


And if you hit OK, you will find that our server will be started up and you will be able to access the server as you did before.



With this setup, you can use the embedded Tomcat server as well. We go to localhost:8080.

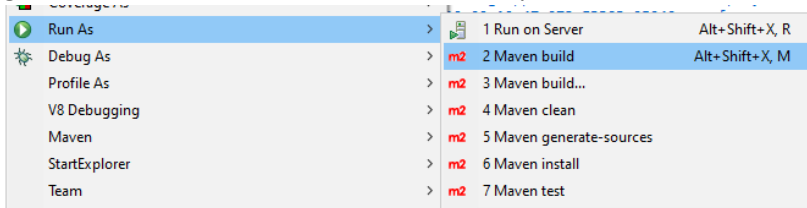
Spring Boot Microservices: Getting Started



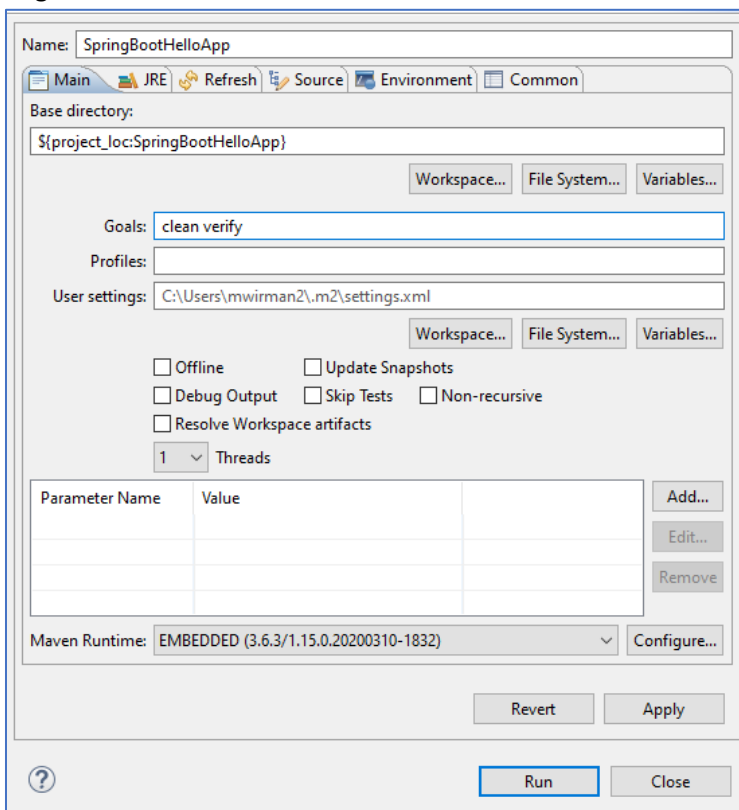
Our WAR deployment works with the embedded server.

5. Build WAR using Maven

Let's actually use an actual external server now. Make sure that you stop the current server that's running. Instead of building this application as a WAR file, select the project. Right-click, go to Run As and choose the Maven build option.

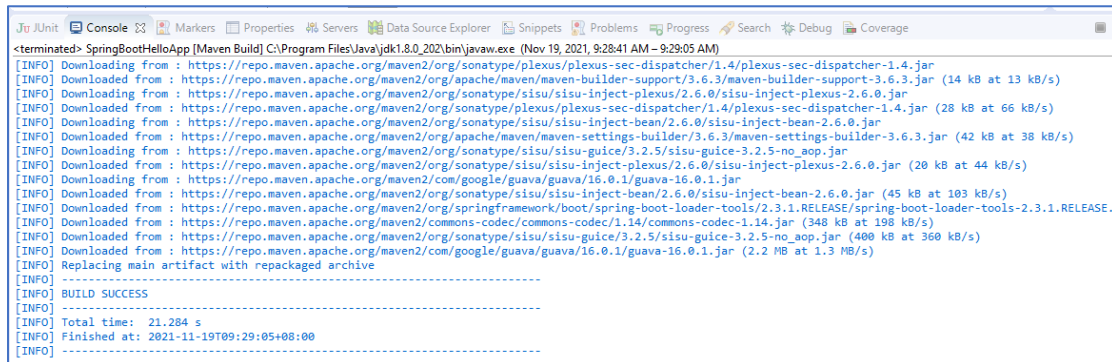


I'll use Maven build to generate a WAR file. Within the goals of this dialogue, specify **clean verify** to get a clean Maven build.



Spring Boot Microservices: Getting Started

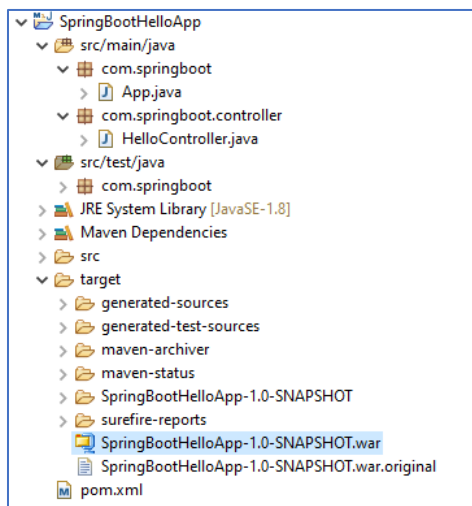
Click on the Run button, and a new WAR file will be generated within your project. This BUILD SUCCESS should tell you that we now have a WAR file, it's time for us to deploy this WAR file to an external server.



```
<terminated> SpringBootHelloApp [Maven Build] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 19, 2021, 9:28:41 AM - 9:29:05 AM)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/sonatype/plexus/plexus-sec-dispatcher/1.4/plexus-sec-dispatcher-1.4.jar
[INFO] Downloaded from : https://repo.maven.apache.org/maven2/org/sonatype/plexus/plexus-sec-dispatcher/1.4/plexus-sec-dispatcher-1.4.jar (14 kB at 13 kB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/apache/maven/maven-builder-support/3.6.3/maven-builder-support-3.6.3.jar
[INFO] Downloaded from : https://repo.maven.apache.org/maven2/org/apache/maven/maven-builder-support/3.6.3/maven-builder-support-3.6.3.jar (28 kB at 66 kB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/sonatype/sisu/sisu-inject-plexus/2.6.0/sisu-inject-plexus-2.6.0.jar
[INFO] Downloaded from : https://repo.maven.apache.org/maven2/org/sonatype/sisu/sisu-inject-plexus/2.6.0/sisu-inject-plexus-2.6.0.jar (42 kB at 38 kB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/sonatype/sisu/sisu-inject-bean/2.6.0/sisu-inject-bean-2.6.0.jar
[INFO] Downloaded from : https://repo.maven.apache.org/maven2/org/sonatype/sisu/sisu-inject-bean/2.6.0/sisu-inject-bean-2.6.0.jar (20 kB at 44 kB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/com/google/guava/guava/16.0.1/guava-16.0.1.jar
[INFO] Downloaded from : https://repo.maven.apache.org/maven2/com/google/guava/guava/16.0.1/guava-16.0.1.jar (45 kB at 103 kB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/org/sonatype/sisu/sisu-inject-bean/2.6.0/sisu-inject-bean-2.6.0.jar
[INFO] Downloaded from : https://repo.maven.apache.org/maven2/org/sonatype/sisu/sisu-inject-bean/2.6.0/sisu-inject-bean-2.6.0.jar (400 kB at 360 kB/s)
[INFO] Downloading from : https://repo.maven.apache.org/maven2/commons-codec/commons-codec/1.14/commons-codec-1.14.jar
[INFO] Downloaded from : https://repo.maven.apache.org/maven2/commons-codec/commons-codec/1.14/commons-codec-1.14.jar (348 kB at 198 kB/s)
[INFO] Replacing main artifact with repackaged archive
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 21.284 s
[INFO] Finished at: 2021-11-19T09:29:05+08:00
[INFO] -----
```

6. Deploy War into Tomcat

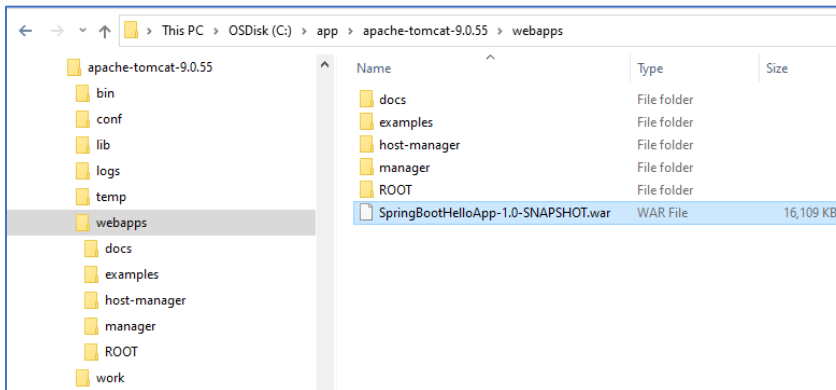
We've already generated a WAR file using the Maven build process, click on the back button and expand the folder for your Eclipse project. Under this folder is where we'll find our WAR file. You should be able to see SpringBootHelloApp-1.0-SNAPSHOT.war.



This is the WAR file that we need to deploy to the webapps folder within the Tomcat server. So expand the Tomcat server and you see the webapps folder there and the WAR file. Select the

Spring Boot Microservices: Getting Started

WAR file and drag it into the webapps folder.



Really, that's all you need to deploy a WAR file on your locally running Tomcat. Go ahead and drop the WAR file into the webapps directory. You've now successfully deployed a WAR file. Notice that SpringBootHelloApp.war is now under webapps under Tomcat server.

7. Start Tomcat Server

With our WAR file deployed, it's now time to start up our server, cd into the bin directory of your Tomcat server.

Here we are within the bin directory. Let's now start up our Tomcat server.

The command that you will use on your Mac machine. You will simply run the startup.sh script and tail the catalina logs.

Windows machine there is a batch file that you can execute, simply run startup.bat within this bin folder, and that will bring up your local Tomcat server.

```
Windows PowerShell
PS C:\app\apache-tomcat-9.0.55\bin> .\startup.bat
Using CATALINA_BASE: "C:\app\apache-tomcat-9.0.55"
Using CATALINA_HOME: "C:\app\apache-tomcat-9.0.55"
Using CATALINA_TMPDIR: "C:\app\apache-tomcat-9.0.55\temp"
Using JRE_HOME: "C:\Program Files\Java\jdk1.8.0_202"
Using CLASSPATH: "C:\app\apache-tomcat-9.0.55\bin\bootstrap.jar;C:\app\apache-tomcat-9.0.55\bin\tomcat-juli.jar"
Using CATALINA_OPTS: ""
```

```
Tomcat
19-Nov-2021 09:50:47.838 INFO [main] org.apache.catalina.startup.HostConfig.deployWAR Deploying web application archive [C:\app\apache-tomcat-9.0.55\webapps\SpringBootHelloApp-1.0-SNAPSHOT.war]
19-Nov-2021 09:50:50.217 INFO [main] org.apache.jasper.servlet.TldScanner.scanJars At least one JAR was scanned for TLDs yet contained no TLDs. Enable debug logging for this logger for a complete list of JARs that were scanned but no TLDs were found in them. Skipping unneeded JARs during scanning can improve startup time and JSP compilation time.

Spring
:: Spring Boot ::
(v2.3.1.RELEASE)

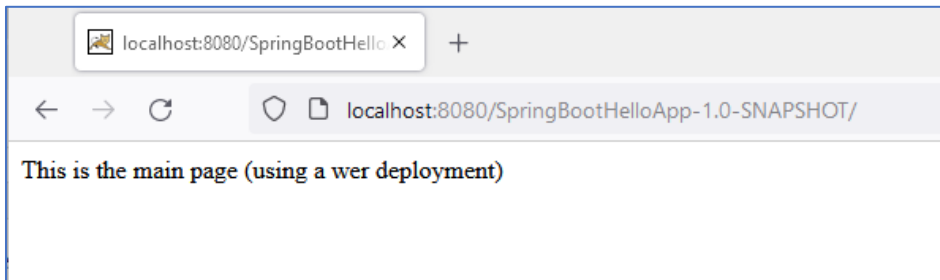
2021-11-19 09:50:51.361 INFO [main] com.springboot.App : Starting App v1.0-SNAPSHOT on US-SCD8144SQ9 with PID 27288 (C:\app\apache-tomcat-9.0.55\webapps\SpringBootHelloApp-1.0-SNAPSHOT\WEB-INF\classes started by mirmann2 in C:\app\apache-tomcat-9.0.55\bin)
2021-11-19 09:50:51.363 INFO [main] com.springboot.App : No active profile set, falling back to default profiles: default
2021-11-19 09:50:52.509 INFO [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1097 ms
2021-11-19 09:50:52.882 INFO [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-11-19 09:50:53.211 INFO [main] com.springboot.App : Started App in 2.637 seconds (JVM running for 6.078)
19-Nov-2021 09:50:54.545 WARNING [main] org.apache.catalina.util.SessionIdGeneratorBase.createSecureRandom Creation of SecureRandom instance for session ID generation using [SHA1PRNG] took [1.319] milliseconds.
19-Nov-2021 09:50:54.561 INFO [main] org.apache.catalina.startup.HostConfig.deployWAR Deployment of web application archive [C:\app\apache-tomcat-9.0.55\webapps\SpringBootHelloApp-1.0-SNAPSHOT.war] has finished in [6.723] ms
19-Nov-2021 09:50:54.561 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [C:\app\apache-tomcat-9.0.55\webapps\docs]
19-Nov-2021 09:50:54.592 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\app\apache-tomcat-9.0.55\webapps\docs] has finished in [31] ms
19-Nov-2021 09:50:54.592 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [C:\app\apache-tomcat-9.0.55\webapps\examples]
19-Nov-2021 09:50:54.905 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\app\apache-tomcat-9.0.55\webapps\examples] has finished in [313] ms
19-Nov-2021 09:50:54.905 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [C:\app\apache-tomcat-9.0.55\webapps\host-manager]
19-Nov-2021 09:50:54.937 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\app\apache-tomcat-9.0.55\webapps\host-manager] has finished in [32] ms
19-Nov-2021 09:50:54.937 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [C:\app\apache-tomcat-9.0.55\webapps\manager]
19-Nov-2021 09:50:54.978 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\app\apache-tomcat-9.0.55\webapps\manager] has finished in [21] ms
19-Nov-2021 09:50:54.978 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [C:\app\apache-tomcat-9.0.55\webapps\ROOT]
19-Nov-2021 09:50:54.978 INFO [main] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\app\apache-tomcat-9.0.55\webapps\ROOT] has finished in [20] ms
19-Nov-2021 09:50:54.978 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]
19-Nov-2021 09:50:54.998 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in [7196] milliseconds
```


Spring Boot Microservices: Getting Started

If you're working on a And that's all you need to do to run the local Tomcat server that we have. This is the Tomcat server that is outside of our Spring Boot application.

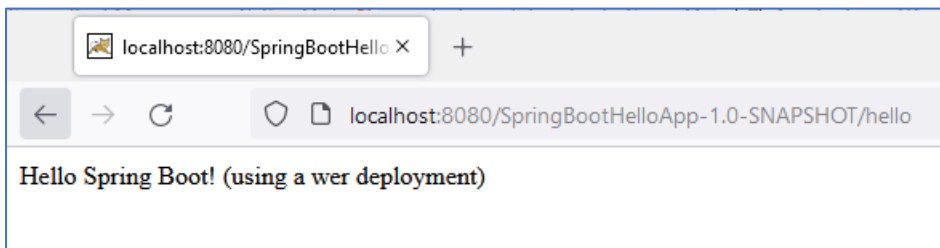
8. Test on Browser

Let's go ahead to our browser window to take a look at our web application that has been deployed. The path to our web application is localhost:8080/SpringBootHelloApp-1.0-SNAPSHOT. In includes the version and the name of the package as well. And you can see displayed the main page,

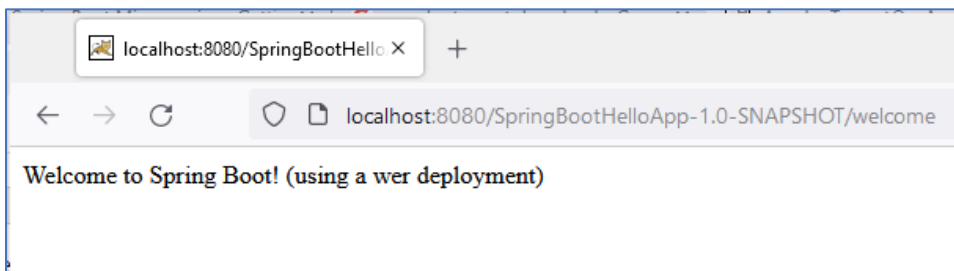


we've successfully deployed our package as a WAR.

Let's take a look at the /hello path and you should see Hello Spring Boot!, once again using a WAR deployment.



And lastly, let's make sure the /welcome path works, and this works as well. You can see Welcome to Spring Boot! printed on screen.



9. Check and Stop Tomcat Service

We've successfully deployed our Spring Boot application using an external WAR file. Go ahead and kill this server, a Ctrl C should take care of this. We'll also ensure that we have nothing listening on port 8080, because we have other servers that we need to run on this port.

Spring Boot Microservices: Getting Started

So I'll run the `lsof -i :8080 | grep LISTEN` command on a Mac OS to see if I have any processes which are listening on this port. If you're on a Windows machine, the command that will give you the process listening on particular port is `netstat -ano | findstr:8080`. Run this command, and you should get the process ID listening on a particular port.

```
Windows PowerShell
PS C:\app\apache-tomcat-9.0.55\bin> .\startup.bat
Using CATALINA_BASE: "C:\app\apache-tomcat-9.0.55"
Using CATALINA_HOME: "C:\app\apache-tomcat-9.0.55"
Using CATALINA_TMPDIR: "C:\app\apache-tomcat-9.0.55\temp"
Using JRE_HOME: "C:\Program Files\Java\jdk1.8.0_202"
Using CLASSPATH: "C:\app\apache-tomcat-9.0.55\bin\bootstrap.jar;C:\app\apache-tomcat-9.0.55\bin\tomcat-juli.jar"
Using CATALINA_OPTS: ""
PS C:\app\apache-tomcat-9.0.55\bin> netstat -ano | findstr :8080
TCP        0.0.0.0:8080          0.0.0.0:0             LISTENING       10336
TCP        127.0.0.1:53962       127.0.0.1:8080        TIME_WAIT       0
TCP        [::]:8080            [::]:0                LISTENING       10336
TCP        [2001:e68:5411:a1c:50b4:b64:283b:e74]:53998 [2001:e68:5411:a1c:50b4:b64:283b:e74]:8080 TIME_WAIT       0
PS C:\app\apache-tomcat-9.0.55\bin>
```

My process ID happens to be [10336](#).

I'll now kill this process so that port 8080 is free.

On a Mac OS I use the `sudo kill -9` command to force kill this process. If you're on a Windows machine, you can use the `taskkill` command from your dos shell, `taskkill /pid 31357`.

```
PS C:\app\apache-tomcat-9.0.55\bin> netstat -ano | findstr :8080
TCP        0.0.0.0:8080          0.0.0.0:0             LISTENING       10336
TCP        127.0.0.1:53962       127.0.0.1:8080        TIME_WAIT       0
TCP        [::]:8080            [::]:0                LISTENING       10336
TCP        [2001:e68:5411:a1c:50b4:b64:283b:e74]:53998 [2001:e68:5411:a1c:50b4:b64:283b:e74]:8080 TIME_WAIT       0
PS C:\app\apache-tomcat-9.0.55\bin> taskkill /pid 10336
SUCCESS: Sent termination signal to the process with PID 10336.
PS C:\app\apache-tomcat-9.0.55\bin> netstat -ano | findstr :8080
PS C:\app\apache-tomcat-9.0.55\bin>
```

Go ahead and kill this process, you're ready to go back to working on Spring Boot.

Spring Boot Microservices: Getting Started

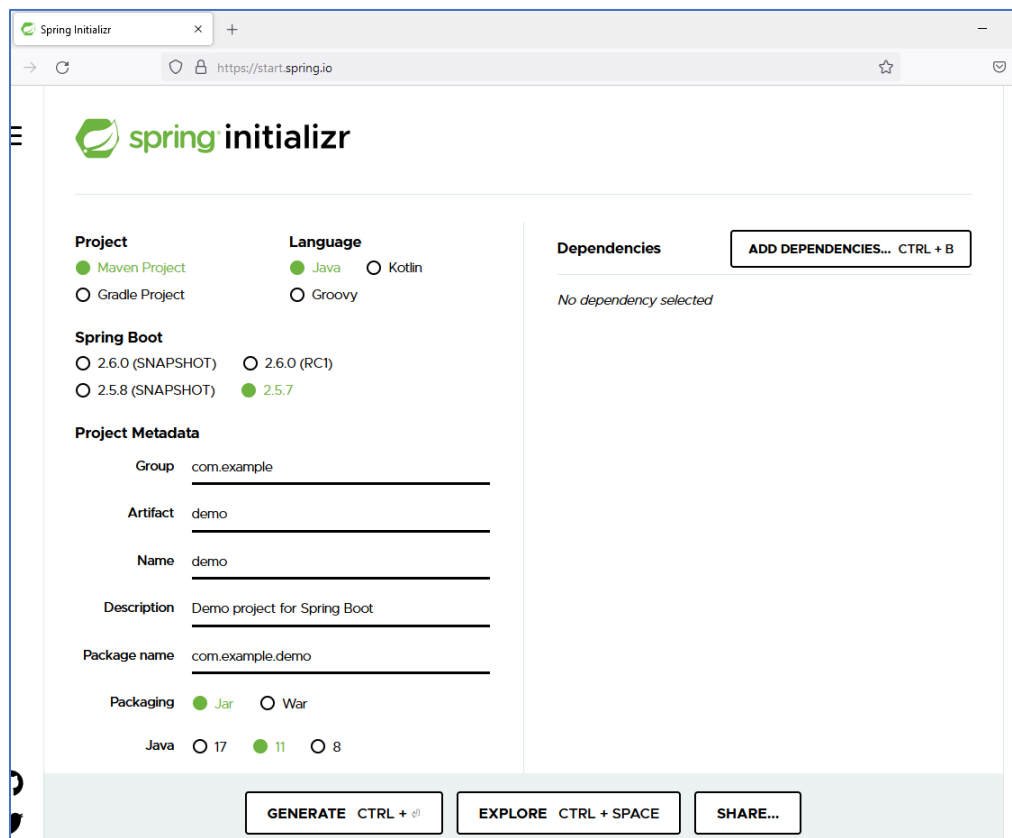
Using Spring Initializr

At this point in time, you're somewhat familiar with how you can set up your Spring Boot projects. So far we've only worked with very simple web applications, but there are so many different microservices that you could build using Spring Boot. In all of these cases, how do you know what the right starter dependencies are? And how do you know you've set up the basic structure of your project right? And that's exactly where Spring Initializer helps us.

Spring Initializer is a **web-based tool** which can be used to generate the basic project structure for Spring Boot. The Spring Initializer web interface is maintained by Pivotal Software. These are the same developers that maintain Spring Boot and the Spring framework.

1. [Get Spring Boot Template Project from http://start.spring.io](http://start.spring.io)

Let's head over to **start.spring.io** and take a look at the Spring Initializer web interface. Remember, this is not going to write any code for you, only set up your project structure and dependencies.



The screenshot shows the Spring Initializr web interface in a browser window. The URL is <https://start.spring.io>. The interface is divided into several sections:

- Project:** ☒ Maven Project, ☐ Gradle Project
- Language:** ☒ Java, ☐ Kotlin, ☐ Groovy
- Spring Boot:** ☐ 2.6.0 (SNAPSHOT), ☐ 2.6.0 (RC1), ☐ 2.5.8 (SNAPSHOT), ☒ 2.5.7
- Project Metadata:**
 - Group:
 - Artifact:
 - Name:
 - Description:
 - Package name:
- Packaging:** ☒ Jar, ☐ War
- Java:** ☐ 17, ☒ 11, ☐ 8
- Dependencies:** . Below it, it says "No dependency selected".

At the bottom, there are three buttons: , , and

On the left side, you specify the basic setup of your project. It's a Maven project.

The language that we're using for development is Java. Once these basic specifications are complete, you can specify the version of Spring Boot that you want to use. We are using version **2.5.7**, the latest at the time of this recording.

On the right side is where we specify the dependencies that we want to use for our project. But before we get to that, let's scroll down below and see how we can configure the Project

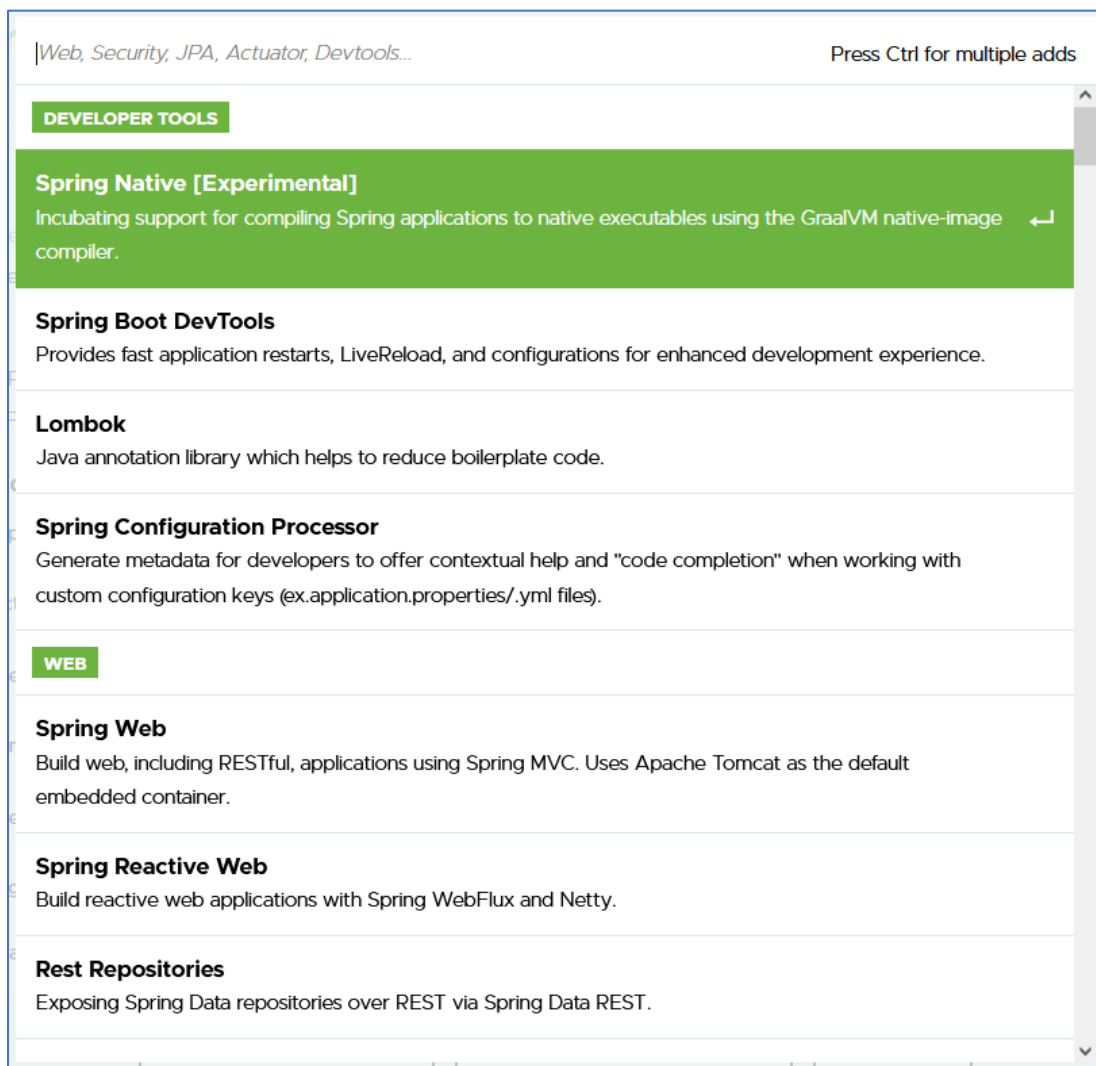
Spring Boot Microservices: Getting Started

Metadata. In the Project Metadata section is where we specify the group, the artifact ID, the packaging for our application and all of the other details.

The group that I'll use here is *com.springboot*. Use the reverse domain name of your organization. The artifact that I want for this project is simply *springinitializer*, indicating that I've initialize this project using this web interface. You can specify a more meaningful description for your project if you want to. You can see that the package name has automatically been generated, *com.springboot.springinitializer*.

We want to generate a Jar file, that is the packaging option that is checked. And the Java version that we want our application to be built on is Java 8. You can choose Java 11 or 14 if you have that installed. Let's scroll back up and specify the dependencies for this project. Off to the top right you'll find the ADD DEPENDENCIES button. And this will bring up a dialog, giving you a list of all possible Spring dependencies that you can choose from.

There is also a search bar at the very top allowing you to search for dependencies.



Spring Boot Microservices: Getting Started

You can take a quick look at what is available by scrolling down. Here is Spring Session for session-based applications, you have TEMPLATE ENGINES, you have security-based applications. And if you want to connect to a SQL back end, you have SQL-based dependencies as well such as JDBC, JPA, Hibernate, and so on. If you're working on a Spring-based application for some kind of cloud platform, you have cloud-based dependencies available as well. Such as for Amazon Web Services and Microsoft Azure.

Since we're just getting started with Spring at the moment, I'm going to scroll to the top. And under the WEB section, I'm going to select the Spring Web dependency. Select Spring Web and this will be added within your Dependencies section.

The screenshot shows the Spring Initializr web application. At the top is the 'spring initializr' logo. Below it, there are several sections for configuring a project:

- Project:** Radio buttons for 'Maven Project' (selected) and 'Gradle Project'.
- Language:** Radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'.
- Spring Boot:** Radio buttons for versions '2.6.0 (SNAPSHOT)', '2.6.0 (RC1)', '2.5.8 (SNAPSHOT)', and '2.5.7' (selected).
- Project Metadata:** Fields for 'Group' (com.springboot), 'Artifact' (springinitializer), 'Name' (springinitializer), 'Description' (Demo project for Spring Boot), and 'Package name' (com.springboot.springinitializer).
- Packaging:** Radio buttons for 'Jar' (selected) and 'War'.
- Java:** Radio buttons for versions '17', '11', and '8' (selected).
- Dependencies:** A section with a button 'ADD DEPENDENCIES... CTRL + B'. Below it, 'Spring Web' is selected with a 'WEB' tag. A description reads: 'Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.'

At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

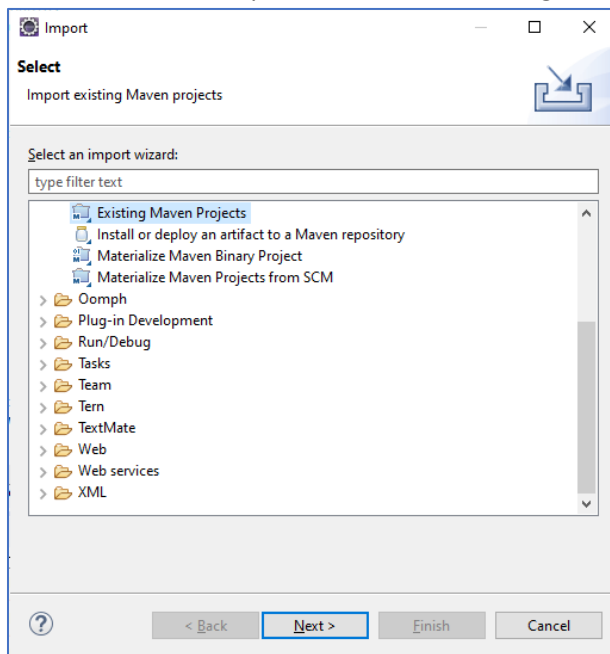
Now all you need to do is click on the GENERATE button here and a starter project will be generated for you and downloaded as a ZIP file.

I'm going to take a look at where this ZIP file is located within my Finder window. Move this ZIP file to where your project's folder is. I've placed this ZIP file here next to my other projects in my current working directory. I'm going to double-click on this ZIP file, extract all of the contents and this springinitializer folder will contain the basic structure of my app.

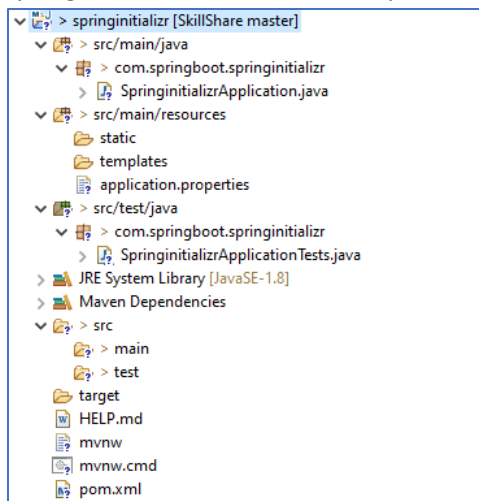
Spring Boot Microservices: Getting Started

2. Import Maven Project in Eclipse

It's now time for us to head over to Eclipse and import this Maven project as we've done before. Go to File, Import, and from this dialog here, choose Existing Maven Projects.



Click on the Next button, you will be asked to specify the directory where your Maven project exists. Click on the Browse button and browse to your Spring Boot projects folder. And within there, I have the springinitializer directory. This is the project structure set up using the springinitializer web interface, open this folder and click on Finish.



And you can see within this springinitializer project, we have the entire structure of a Spring Boot application set up for us.

- pom.xml

Let's take a look at the pom.xml and you'll see that it has been configured using all of the dependencies that we have specified.

Spring Boot Microservices: Getting Started

```
springinitializr/pom.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.springboot</groupId>
12  <artifactId>springinitializr</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springinitializr</name>
15  <description>Demo project for Spring Boot</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter-web</artifactId>
23    </dependency>
24
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-test</artifactId>
28      <scope>test</scope>
29    </dependency>
30  </dependencies>
31
32  <build>
33    <plugins>
34      <plugin>
35        <groupId>org.springframework.boot</groupId>
36        <artifactId>spring-boot-maven-plugin</artifactId>
37      </plugin>
38    </plugins>
39  </build>
40
41 </project>
```

There on line 7 is the spring-boot-starter-parent,

```
<artifactId>spring-boot-starter-parent</artifactId>
```

the pom that we inherit from for all of the Spring Boot dependencies. And if you scroll down, you will see that on line 20, we have the spring-boot-starter-web dependency descriptor.

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

In addition, for every Spring project we have the spring-boot-starter-test dependency descriptor as well.

You can see this on line 25.

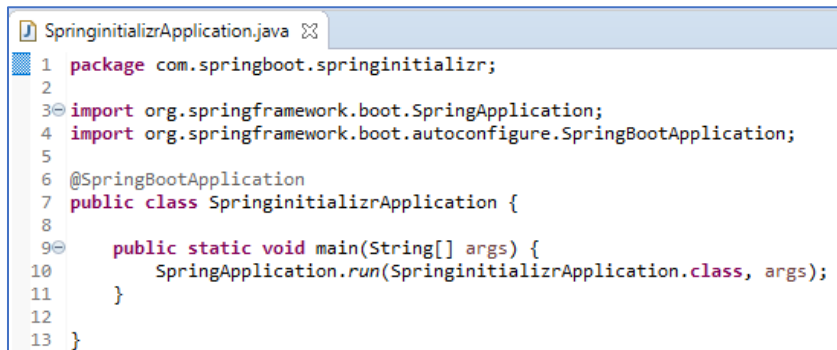
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Spring Boot Microservices: Getting Started

If you scroll further down, you'll see that this generated project also includes the spring-boot-maven-plugin.

- Default Entry Point Application (SpringinitializerApplication)

If you head over to the src/main/java folder, you can see that the entry point of our Spring Boot application has also been set up for us. The default name of this class is SpringinitializerApplication, but you can of course change this.

A screenshot of a code editor showing the file SpringinitializerApplication.java. The code is as follows:

```
1 package com.springboot.springinitializr;  
2  
3 import org.springframework.boot.SpringApplication;  
4 import org.springframework.boot.autoconfigure.SpringBootApplication;  
5  
6 @SpringBootApplication  
7 public class SpringinitializrApplication {  
8  
9     public static void main(String[] args) {  
10         SpringApplication.run(SpringinitializrApplication.class, args);  
11     }  
12  
13 }
```

Spring Initializer has set up the @SpringBootApplication annotation on this class and it has a main method as well which simply runs this class.

3. Create new Controller Class

Let's set up the same web application that we saw in a previous demo. I'm going to create a new package here within my src/main/java folder, right-click, click on New and go to Package. I'm going to have my controller class be in a separate package, the name of the package is *com.springboot.springinitializer.controller*. Once the package has been created. I'm going to right-click on that package, go to New and Class to generate a new class file for my controller.

This will be the *HelloController* that we've seen before. Click on Finish and a new file will be generated. Let's just paste the code that we had from our previous demo into this file.

Spring Boot Microservices: Getting Started

```
1 package com.springboot.springinitializr.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 public class HelloController {
8
9     @GetMapping("/")
10    public String index() {
11        return "We've successfully used the Spring initializr";
12    }
13
14    @GetMapping("/welcome")
15    public String welcome() {
16        return "Welcome to Spring Boot!";
17    }
18
19    @GetMapping("/hello")
20    public String hello() {
21        return "Hello Spring Boot!";
22    }
23
24 }
```

This HelloController will be a simple RestController, you can see the @RestController annotation on the class with three handler mappings.

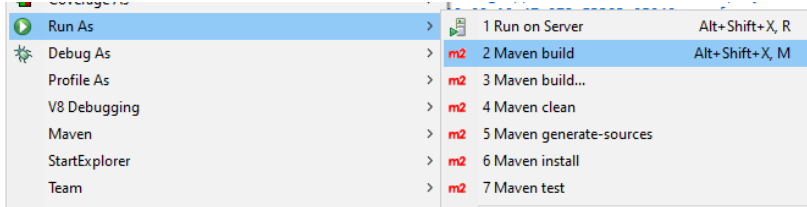
We have the / mapping for the main page, where we say we've successfully used the Spring Initializer. We have a mapping for /welcome and /hello as well.

All of these handler methods respond to GET request. Notice how I've tagged these methods. Instead of using the @RequestMapping annotation, I've used the @GetMapping annotation. The @GetMapping allows us to eliminate the extra parameter indicating the type of request for this method. All three of these handler mappings are for HTTP GET requests.

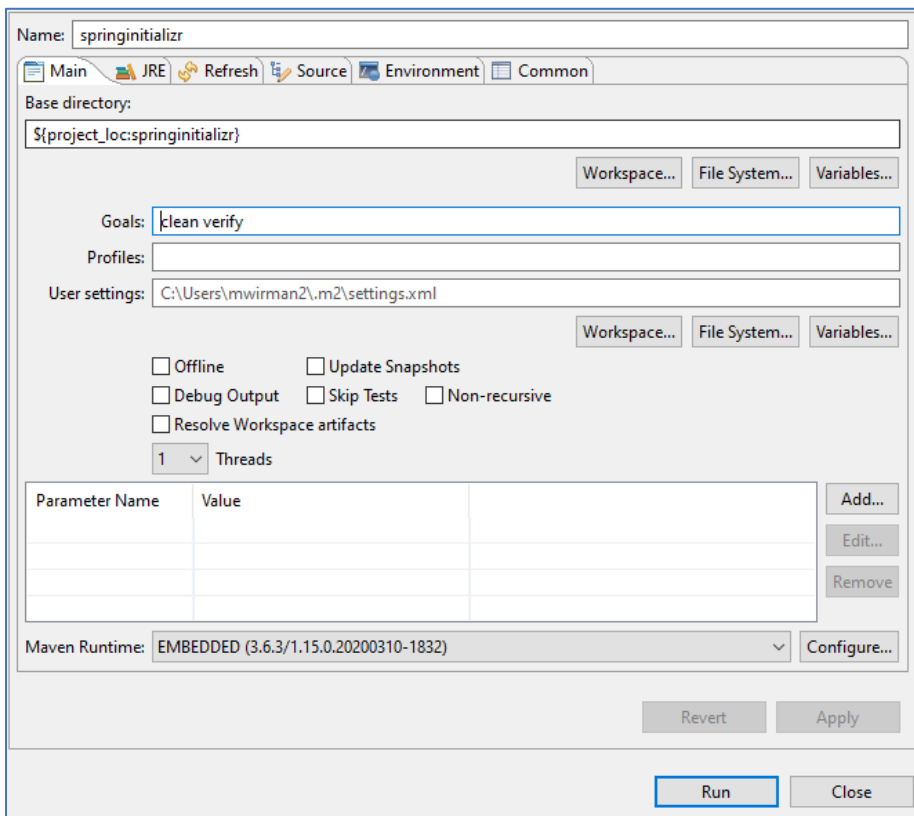
Spring Boot Microservices: Getting Started

4. Build Maven Project

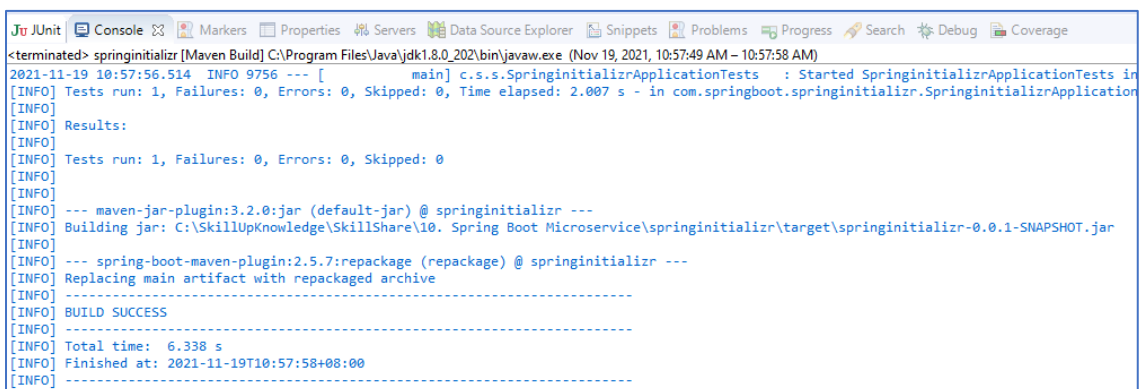
Select the project. Right-click, go to Run As and choose the Maven build option.



I'll use Maven build within the goals of this dialogue, specify **clean verify** to get a clean Maven build.



Click on the Run button.

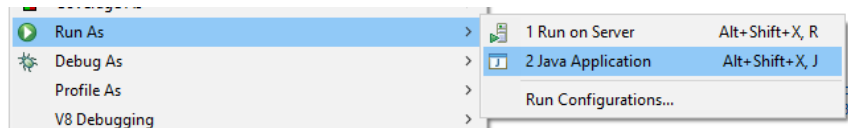


Spring Boot Microservices: Getting Started

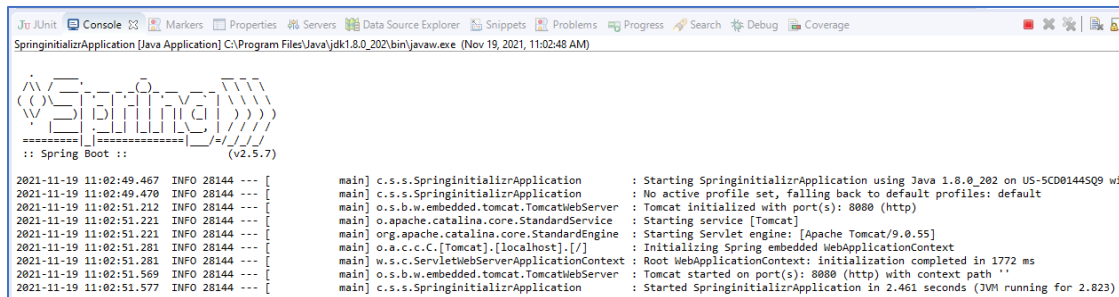
5. Run and Test the Application

Let's run this code to make sure everything works.

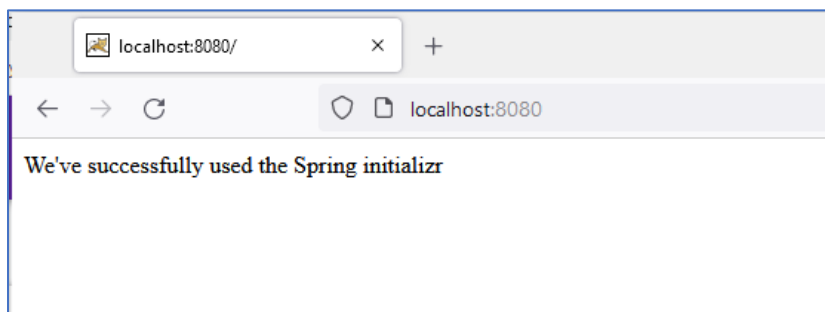
We've used the Spring Initializer to set up the basic project, we wrote the code ourselves. I'm going to right-click on the class **SpringinitializerApplication.java**, choose Run As and Java Application. Select the Spring Initializer application and click on OK.



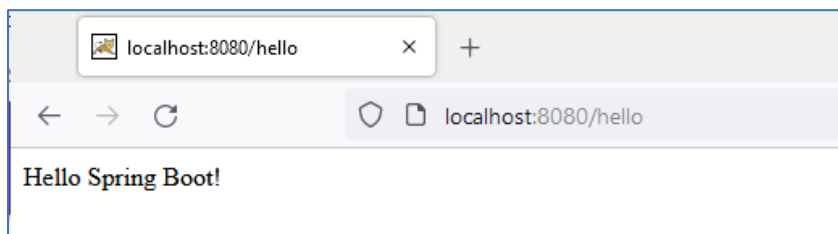
And you'll find that our app runs on the embedded Tomcat server, exactly as it did earlier.



This app should be available for us to access on port 8080. So let's head over to our browser window and hit *localhost:8080*. And you should see we've successfully used the Spring Initializer, and

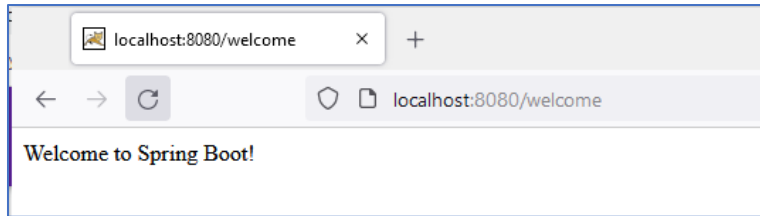


indeed we have. We'll now try a different path in our application here, */hello* and this should say, Hello Spring Boot!



There is one last mapping left to try, to the */welcome* path, and this should say Welcome to Spring Boot!.

Spring Boot Microservices: Getting Started



Performing Message Internationalization

Any web application that we build today, we build for an international audience. So it's quite possible that you want your messages and all of the text in your web applications to be locale specific. In France, you want them to be displayed in French. In India, maybe you want them displayed in Hindi. In this demo, we'll see how you can configure different messages in your web application based on the locale.

1. Update Dependency POM XML

Now here is the **pom.xml**.

```
springinitializr/pom.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.springboot</groupId>
12  <artifactId>springinitializr</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springinitializr</name>
15  <description>Demo project for Spring Boot</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter-web</artifactId>
23    </dependency>
24
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-thymeleaf</artifactId>
28    </dependency>
29
30    <dependency>
31      <groupId>org.springframework.boot</groupId>
32      <artifactId>spring-boot-starter-test</artifactId>
33      <scope>test</scope>
34    </dependency>
35  </dependencies>
36
37  <build>
38    <plugins>
39      <plugin>
40        <groupId>org.springframework.boot</groupId>
41        <artifactId>spring-boot-maven-plugin</artifactId>
42      </plugin>
43    </plugins>
44  </build>
45
46 </project>
```

We'll use the spring-boot-starter-web dependency descriptor. This is on line 24.

Spring Boot Microservices: Getting Started

In addition, we'll use the **Thymeleaf** template engine in Spring Boot. The Thymeleaf starter dependency is specified on line 25.

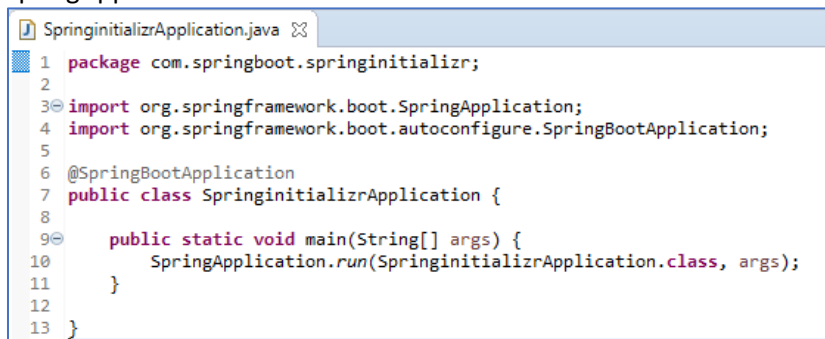
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

So what exactly is Thymeleaf? Thymeleaf is a Java-based template engine typically used in Spring Boot applications used to transform and render the data that you want displayed in your web apps. Thymeleaf is a server-side template engine and it provides great support for rendering HTML5.

Using the Thymeleaf starter template as we've done here automatically sets up a view resolver for our Spring Boot web application. It also allows us to use message properties configured in special files.

2. Entry Point Default Entry Point Application (SpringinitializerApplication)

There is no change in the code for the main entry point of our application. Here is the SpringinitializerApplication file annotated using @SpringBootApplication, which simply calls SpringApplication.run on this class.



```
SpringinitializerApplication.java
1 package com.springboot.springinitializer;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class SpringinitializerApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringinitializerApplication.class, args);
11     }
12
13 }
```

3. Create New Controller Class

This application is fairly simple. It has a single controller class called the MessageController, which is annotated using the @Controller annotation.

MessageController.java



```
MessageController.java
1 package com.springboot.springinitializer.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.GetMapping;
5
6 @Controller
7 public class MessageController {
8
9     @GetMapping("/home")
10     public String index() {
11         return "home";
12     }
13 }
```

The **@Controller** annotation tags this class as one, which handles incoming web request mapped to specific parts. However, the value returned by the handler methods is **NOT** considered to be a **web response**. Instead, the return value of these methods is interpreted as a

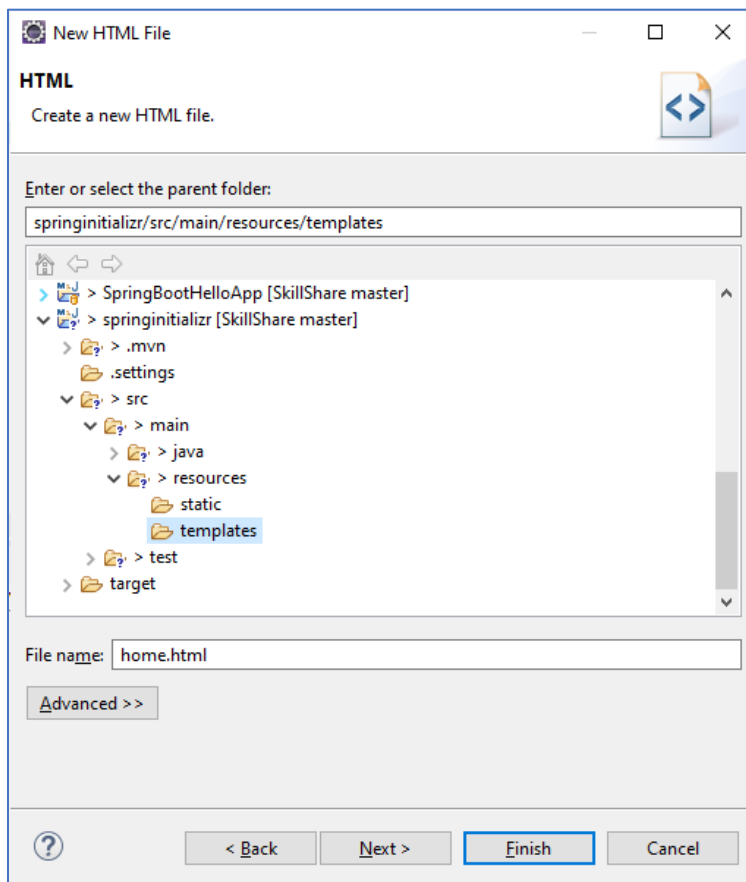
Spring Boot Microservices: Getting Started

logical view name. The Thymeleaf template engine will then map this view name to a physical view which is then rendered by the browser.

We have exactly one mapping specified here within this MessageController class. We have a GetMapping for the path /home on the index function.

4. Create new HTML File under src/main/resources/template folder

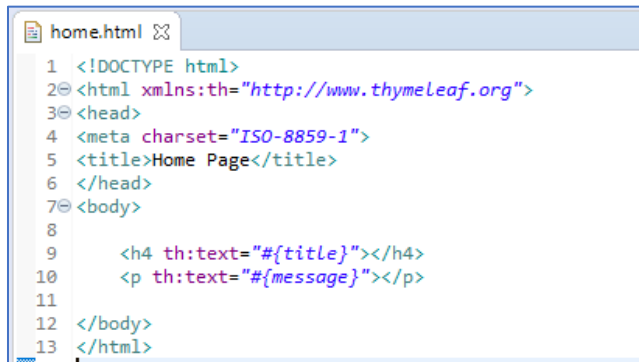
Notice that the index function in MessageController.java simply returns the string **home**. The Thymeleaf template engine will interpret this string as a logical view. It'll look up the physically view corresponding to the string and render that physical view in the form of an HTML file or some other file within our web browser. The physical view corresponding to this logical view name home we place within the templates folder under **src/main/resources**. Expand the **templates** folder here and under that you can see a home.html file. This is the physical rendering of the view home.



And if you look at the contents of this HTML file, you'll see that it's not all plain HTML. Some of it is indeed HTML. But we also have some Thymeleaf specific code in here. This is a view rendered using the Thymeleaf template engine.

Spring Boot Microservices: Getting Started

home.html



```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4 <meta charset="ISO-8859-1">
5 <title>Home Page</title>
6 </head>
7 <body>
8
9 <h4 th:text="#{title}"></h4>
10 <p th:text="#{message}"></p>
11
12 </body>
13 </html>
```

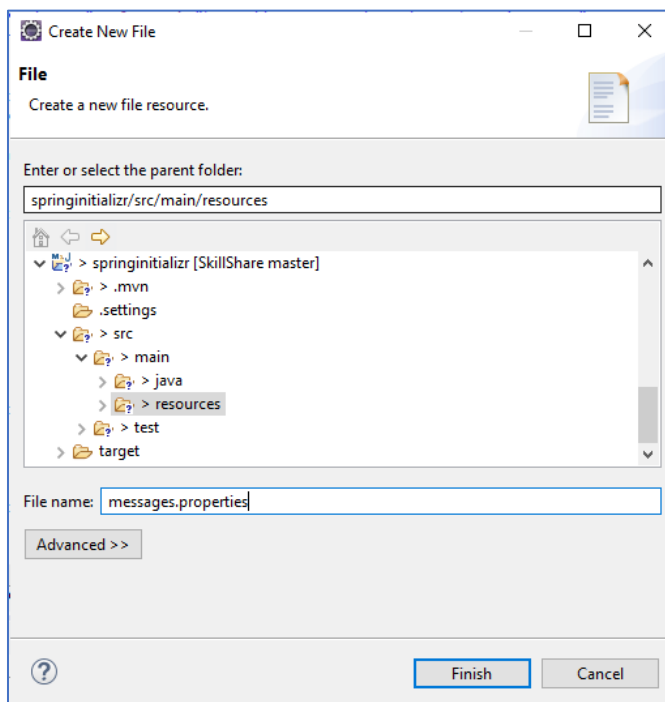
On line 2, you can see we have a reference to www.thymeleaf.org.

```
<html xmlns:th="http://www.thymeleaf.org">
```

And on lines 9 and 10, notice that we use the `th:text` attribute in order to reference the messages that will be displayed on this `home.html` page.

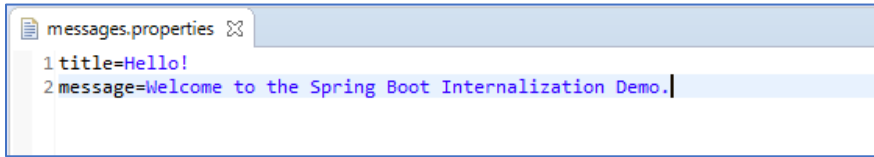
```
<h4 th:text="#{title}"></h4>
<p th:text="#{message}"></p>
```

We'll display the values in the title variable and the message variable. But where exactly are these variables defined? These variables are defined in our **messages.properties** file, which we'll create under resources. Right-click, go to New and select Other, and choose a simple file. Under the General category here in this wizard, you'll find the File option. Select this option and click on Next. Here we'll specify the name of the file. It should be called **messages.properties**.



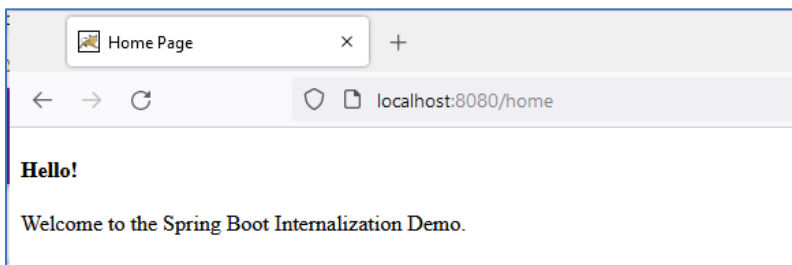
Spring Boot Microservices: Getting Started

Thymeleaf will look for this messages.properties file in order to look up the variables that you've referenced within your template. The messages.properties file gives us the messages using the default locale. We've assumed that English US is the default locale. Title says Hello! The message says, Welcome to the Spring Boot Internationalization Demo. We haven't configured messages in other locales yet.

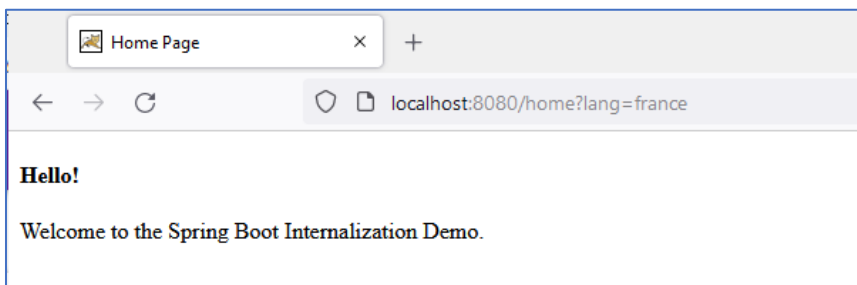


```
messages.properties
1 title=Hello!
2 message=Welcome to the Spring Boot Internationalization Demo.
```

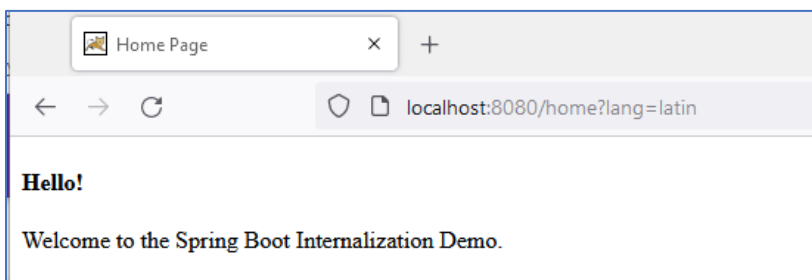
Go ahead build and run this code as a simple application. Let's head over to localhost:8080/home. And you can see the home.html page has been rendered out in English.



For our pages in different locales, we'll use a request parameter. I'm going to use the request param language equal to French. And if I hit Enter, because I haven't configured any international messages yet, I still see the same page in English.



Let's try language equal to Latin. Once again, I see the page in English.



We haven't configured any message properties for these other languages.

Spring Boot Microservices: Getting Started

5. Create New Service Class to Handle Internalization

Back to our Eclipse IDE, let's fix the internationalization issue. Now under `src/main/java`, I'm going to create a new package called **`com.springboot.springinitializr.config`**. And within this new config package, I'm going to create a new class. This is a class that will hold some additional configuration for the internationalization of my messages. **InternationalizationService** is the name of this class.

Once this class file has been created, I'm going to paste in a bunch of code that will allow me to display messages in different languages based on the `lang` parameter that I pass in with my request.

InternationalizationService.java

```
InternationalizationService.java
1 package com.springboot.springinitializr.config;
2
3 import java.util.Locale;
4
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.web.servlet.LocaleResolver;
8 import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
9 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
10 import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
11 import org.springframework.web.servlet.i18n.SessionLocaleResolver;
12
13 @Configuration
14 public class InternationalizationService implements WebMvcConfigurer {
15
16     @Bean
17     public LocaleResolver localeResolver() {
18         SessionLocaleResolver localeResolver = new SessionLocaleResolver();
19         localeResolver.setDefaultLocale(Locale.US);
20         return localeResolver;
21     }
22
23     @Bean
24     public LocaleChangeInterceptor localeChangeInterceptor() {
25         LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
26         localeChangeInterceptor.setParamName("lang");
27         return localeChangeInterceptor;
28     }
29
30     @Override
31     public void addInterceptors(InterceptorRegistry registry) {
32         registry.addInterceptor(localeChangeInterceptor());
33     }
34
35
36 }
```

Notice that this `InternationalizationService` class implements the **WebMvcConfigurer** interface. If you want to specify Java-based configuration for your Spring MVC projects, you need your configuration class to implement this **WebMvcConfigurer** interface. We'll tag this class using the **@Configuration** annotation.

The use of the **@Configuration** annotation indicates that a class declares one or more bean methods and may be processed by the Spring container to generate these bean definitions. If you scroll down a bit, you'll see all of the beans that we have defined in order to support the internationalization of our web page.

Spring Boot Microservices: Getting Started

- **public** LocaleResolver localeResolver()
The first bean here on line 17 is the localeResolver. This localeResolver bean allows us to determine the default locale for our application. We've set the default locale to be Locale.US. The default language for our application will be US English.

```
localeResolver.setDefaultLocale(Locale.US);
```

We then set up a localeChangeInterceptor.

- **public** LocaleChangeInterceptor localeChangeInterceptor()
This is on line 24. An interceptor in Spring is a component that processes your request before passing your request on to Spring MVC's controller. All of your incoming web requests along with the request path will pass through this localeChangeInterceptor. It will be processed before it's passed on to the controller that renders the view corresponding to that path.

This localeChangeInterceptor basically sets the ParamName lang. This is the request parameter that will allow us to determine the locale or the language in which we want our web app to be displayed.

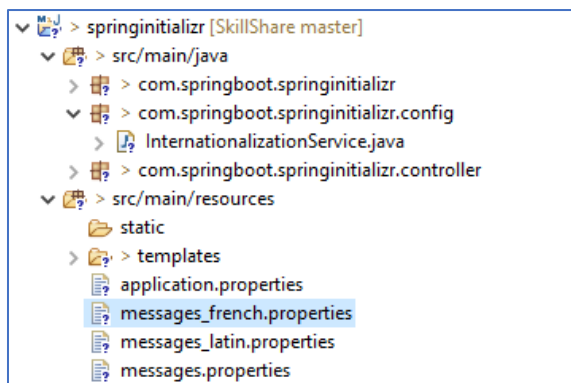
```
localeChangeInterceptor.setParamName("lang");
```

- **public void** addInterceptors(InterceptorRegistry registry)
Now in order for this localeChangeInterceptor to actually take effect, we need to explicitly add this interceptor to the interceptor registry in our application, which we do on line 32. Using this interceptor essentially tells Spring use the *lang* request param in order to determine the locale of this app and the language in which the messages will be displayed.

```
registry.addInterceptor(localeChangeInterceptor());
```

6. Create Mode Properties files for another Locale Language

Now, we need to set up the messages in different languages. And I do this by adding additional *.properties* file under *src/main/resources*. If you look at *src/main/resources* on the left navigation pane, you'll see files for **messages_french.properties** and **messages_latin.properties**. The naming of these files is important. Notice that we have **messages_** and then the **locale**, then **.properties**.



Spring Boot Microservices: Getting Started

For French, it is `messages_french.properties`. And you can see that the title and message variables are now in French.

```
messages_french.properties
1 title=Salut!
2 message=Bienvenue dans la d  mo d'internalisation Spring Boot.
```

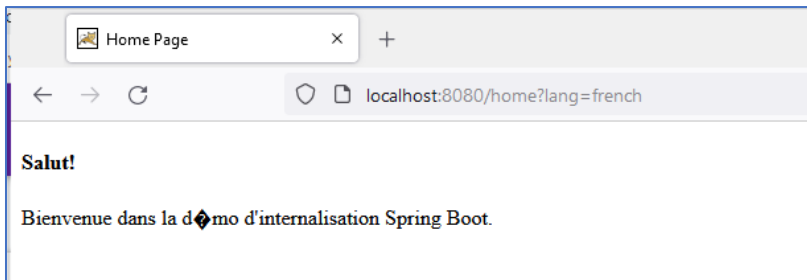
The language or the locale French will be specified using the `lang` request parameter in the incoming request. Similarly, we have a `messages_latin.properties`, where the title and message variables are now in Latin.

```
messages_latin.properties
1 title=Salve!
2 message=Intern ad Spring Boot grata Demo Application.
```

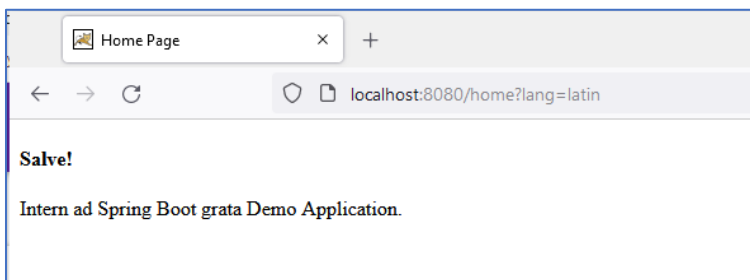
In order to be able to display these messages, our `lang` request parameter has to be set to Latin. Now, this is all the configuration we need to do.

Run your application and let's head over to `localhost:8080/home`.

The URL contains no request parameter, which is why the default locale is US, which means the messages are displayed in English. What if we specify `lang=french`? Notice that the messages are now in French.



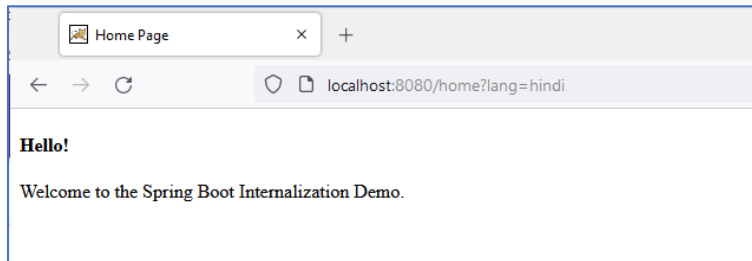
The message properties file which had the `_French` in its name, that's where the messages were picked up from. Let's change the `lang` parameter so that `lang=latin`. Now the messages will be picked up from our Latin file.



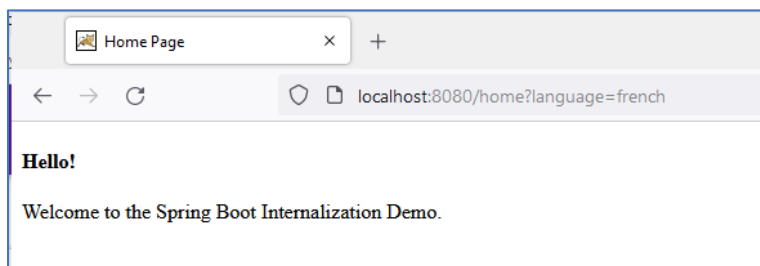
Spring Boot Microservices: Getting Started

What if we specify a value for the lang parameter for which we have no message properties file such as lang=hindi?

In that case, we'll fall back to the default locale and display the messages in English.



What if we specify a different request param *language* instead of *lang*?

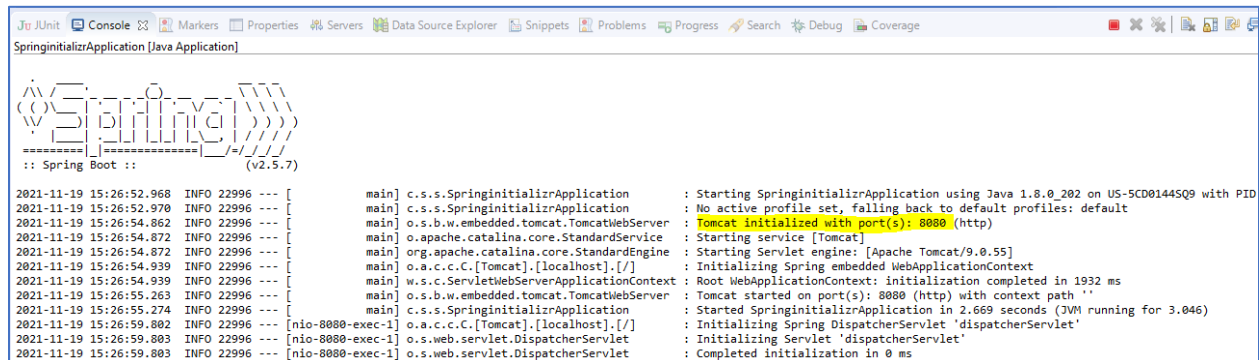


Well, that won't work, because within our configuration, we had specified that the locale resolver param is lang. We have specified in the localeChangeInterceptor that the parameter to intercept is the lang request param. The value of the request parameter must exactly match the name in your message properties file. If you say lang=**fre**, the French properties file will not be picked up. You have to explicitly say **french**.

Spring Boot Microservices: Getting Started

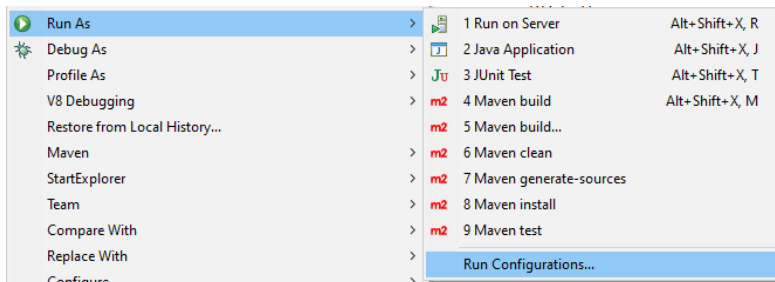
Configuring the Server Port

Let's take a look at how we can change the port on which our web application runs along the way. We know that by default, our server runs on port 8080.

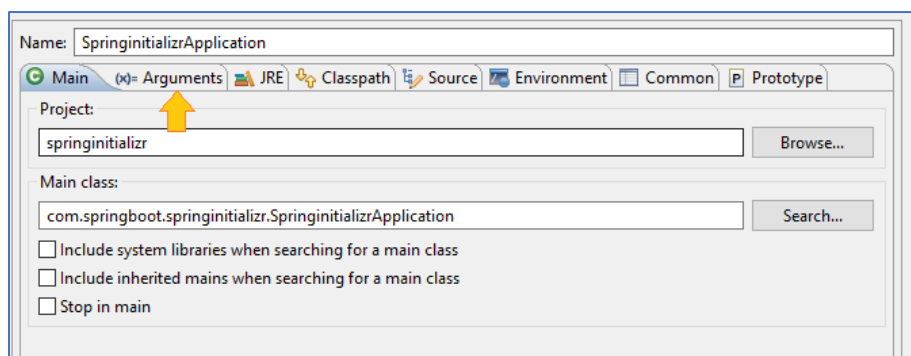


```
2021-11-19 15:26:52.968 INFO 22996 --- [main] c.s.s.SpringinitializrApplication : Starting SpringinitializrApplication using Java 1.8.0_202 on US-5CD0144SQ9 with PID
2021-11-19 15:26:52.970 INFO 22996 --- [main] c.s.s.SpringinitializrApplication : No active profile set, falling back to default profiles: default
2021-11-19 15:26:54.862 INFO 22996 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-19 15:26:54.872 INFO 22996 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-19 15:26:54.872 INFO 22996 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-19 15:26:54.939 INFO 22996 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-19 15:26:54.939 INFO 22996 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1932 ms
2021-11-19 15:26:55.263 INFO 22996 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-19 15:26:55.274 INFO 22996 --- [main] c.s.s.SpringinitializrApplication : Started SpringinitializrApplication in 2.669 seconds (JVM running for 3.046)
2021-11-19 15:26:59.802 INFO 22996 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-11-19 15:26:59.803 INFO 22996 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-11-19 15:26:59.803 INFO 22996 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 0 ms
```

Make sure you stop the currently running server. Now, let's go ahead and select the project, right-click, go to Run As and Run Configurations.

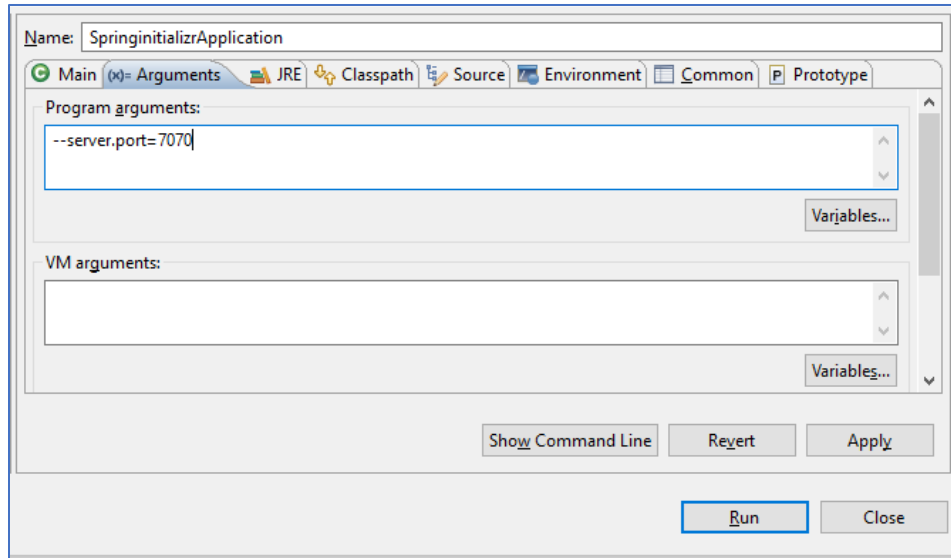


This will bring up all, if you can specify the additional arguments, that you want to pass into our Spring Boot application.

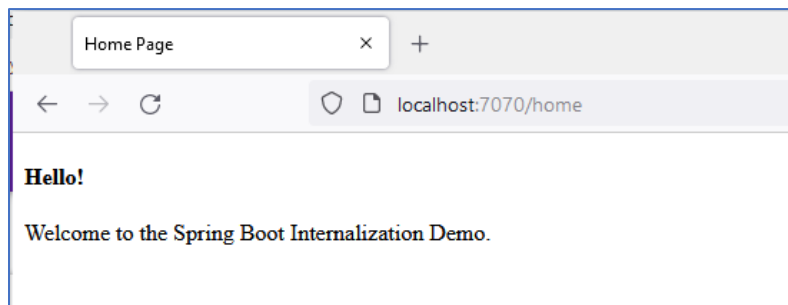


Select the Spring Boot application on the left navigation pane and click on the Arguments tab. This Arguments tab allows us to specify the configuration for our application via command line arguments. I'm going to set `--server.port=7070`, which indicates that our application should run on this port. Click on Apply and then click on the Run button.

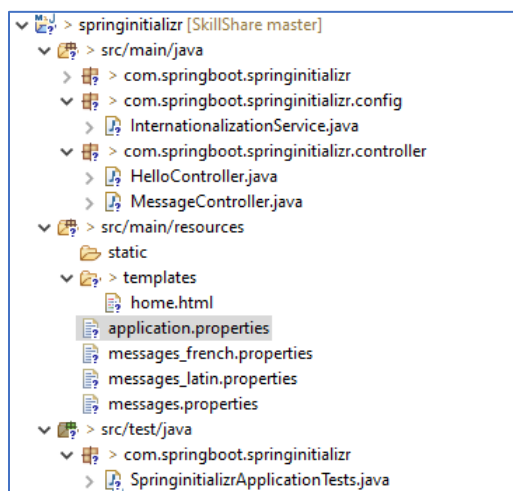
Spring Boot Microservices: Getting Started



Your application will be built and deployed and available on port 7070. If you look at the console messages off to the very right, you'll see that Tomcat has been started on the port 7070 rather than the default 8080. Now, if you head over to our browser and go to localhost:7070, you will find the main page of our app rendered.



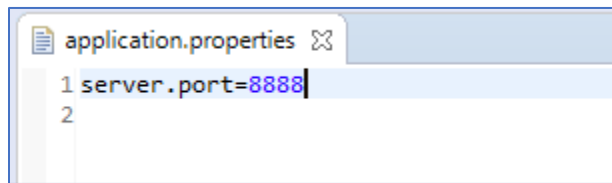
We can also change the port on which our application runs using the application.properties file. Stop the currently running server, and double-click and open up the application.properties file.



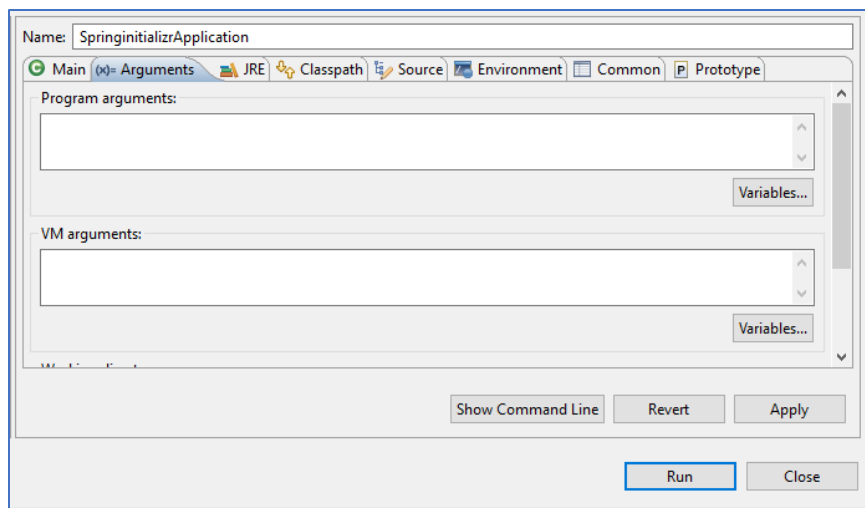
Spring Boot Microservices: Getting Started

This is a simple file where you can specify the properties of your application

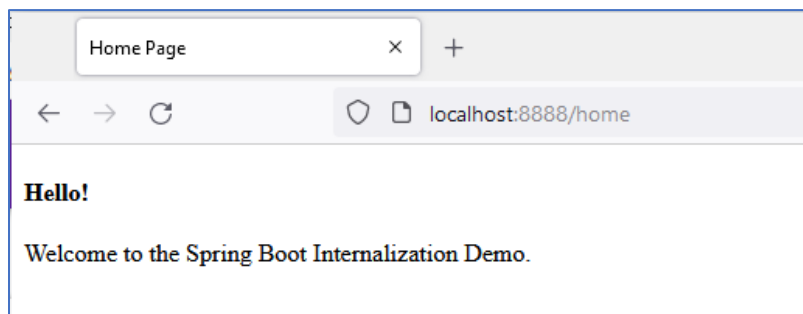
So I've set my server port to 8888.



I'm going to right-click on my project go to Run As and Run Configurations in order to get rid of the extra program argument that I had specified earlier. Head over to the Arguments tab and get rid of the server port input argument present here. Any command line arguments that you specify here will override our specification, the application.properties file.



Once you've gotten rid of this program argument, hit Apply and then hit Run. Your server will be built and your application deployed. And if you take a look at the Tomcat port, you'll see that it's running on port 8888. Time to head over to our browser window and go to localhost:8888. And there you see it, our simple Spring Boot application with two links, the Hello Page and Start Page.



Spring Boot Microservices: Getting Started

Configuring User-defined Properties

In this demo, we'll see how you can specify your own user defined properties in the application.properties file and have Spring automatically identify and pick up these properties as a part of your application.

- Pom.xml

Here is what our **pom.xml** file looks like to start off with. We've included the dependency on **spring-boot-starter-thymeleaf** and **spring-boot-starter-web**.

```
springinitializr/pom.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.springboot</groupId>
12  <artifactId>springinitializr</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springinitializr</name>
15  <description>Demo project for Spring Boot</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter-web</artifactId>
23    </dependency>
24
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-thymeleaf</artifactId>
28    </dependency>
29
30    <dependency>
31      <groupId>org.springframework.boot</groupId>
32      <artifactId>spring-boot-starter-test</artifactId>
33      <scope>test</scope>
34    </dependency>
35  </dependencies>
36
37  <build>
38    <plugins>
39      <plugin>
40        <groupId>org.springframework.boot</groupId>
41        <artifactId>spring-boot-maven-plugin</artifactId>
42      </plugin>
43    </plugins>
44  </build>
45
46 </project>
```

Spring Boot Microservices: Getting Started

- **SpringBoot Entry Point Class**

This is a simple web application where we'll render a few web pages. Let's take a look at our entry point here, the **SpringBootApplication** file.

```
SpringinitializrApplication.java
1 package com.springboot.springinitializr;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class SpringinitializrApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringinitializrApplication.class, args);
11     }
12
13 }
```

There's nothing really that interesting or different so far.

- **Controller Class**

Here's what our controller looks like. We have the GreetingController annotated using the **@Controller** annotation,
GreetingController.java

```
GreetingController.java
1 package com.springboot.springinitializr.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.RequestParam;
7
8 @Controller
9 public class GreetingController {
10
11     @GetMapping("/greeting")
12     public String greeting(@RequestParam String name, Model model) {
13         String message = "Hello " + name;
14         model.addAttribute("greetingMessage", message);
15         return "homePage";
16     }
17 }
```

and we have a single request mapped within it using the **@GetMapping** annotation. This mapping is to the path **/greeting**.

Notice the actual method. This takes in two input arguments, a string **name** and a model. Notice that the String name input argument is annotated using **@RequestParam**.

The **@RequestParam** annotation tells our Spring MVC framework that the value for this name variable will be available as a part of a request parameter made with this request to **/greeting**. The value for that request parameter should be extracted and injected as an input argument to this method invocation. The second input argument, the model, is essentially the model of our MVC application. The model holds the data that we'll pass from the controller to our view.

Spring Boot Microservices: Getting Started

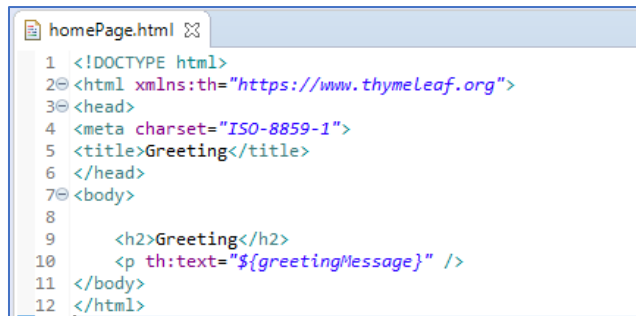
Within this greeting method, we construct a message. The message includes the greeting Hello, and the name passed in as a request parameter.

In order to pass this message from the controller to the view, we use **model.addAttribute**. The key here is *greetingMessage*. The value is the actual message *Hello* and *the name of the person*.

And we return the homepage which is the logical view name that we want rendered.

- Html Page

Let's quickly take a look at the **homePage.html** file present within resources/templates.

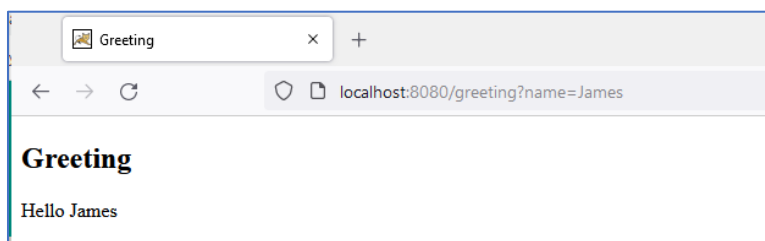


```
1 <!DOCTYPE html>
2 <html xmlns:th="https://www.thymeleaf.org">
3 <head>
4 <meta charset="ISO-8859-1">
5 <title>Greeting</title>
6 </head>
7 <body>
8   <h2>Greeting</h2>
9   <p th:text="${greetingMessage}" />
10 </body>
11 </html>
```

You can see that there is a header here called Greetings! And a simple paragraph element that references the text in our greetingMessage.

Notice how the text is referenced here using \$ and curly braces.

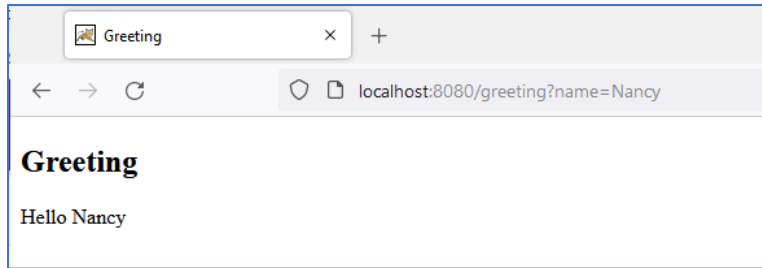
- Run this code and this will bring up our embedded Tomcat server on port 8080. Let's head over to our browser window. We'll hit localhost:8080/greeting. We specify the request parameter name equal to James, and notice that our browser says Hello James.



The **@RequestParam** annotation that we had on the variable name extracts the value associated with the name request parameter which in our case is James. The value James is available in the name variable. In order for this RequestParam injection to work, the name of our variable and the name of the request parameter needs to match exactly.

Let's change the value that we pass in for our name request parameter. I'm going to specify name equal to Nancy. And this time our greeting message will say Hello Nancy.

Spring Boot Microservices: Getting Started



- Configure message with Custom Message Properties
Now, so far, we've been able to configure the name for which we have our greeting. Let's now configure the greeting itself. I'll do this by setting up a separate MessageProperties file within my controller package.

This MessageProperties file I'm going to use to configure the properties of the message that I want displayed. The only configuration property here is the greeting.

```
MessageProperties.java
1 package com.springboot.springinitializr.controller;
2
3 import org.springframework.boot.context.properties.ConfigurationProperties;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 @ConfigurationProperties("messages")
8 public class MessageProperties {
9
10     private String greeting = "Good Morning";
11
12     public String getGreeting() {
13         return greeting;
14     }
15
16     public void setGreeting(String greeting) {
17         this.greeting = greeting;
18     }
19
20 }
```

The MessageProperties class is tagged using the **@Component** annotation, indicating that this should be injected as a bean dependency in our Spring app. I've also tagged it using **@ConfigurationProperties**, and the configuration property is messages.

```
@ConfigurationProperties("messages")
```

The **@ConfigurationProperties** annotation specifies to Spring that this class contains some external configuration properties for our app. And it should be recognized as such. The messages input that we have specified to **@ConfigurationProperties** tells Spring that it should take all properties that start with **messages.**, and try to inject them into this MessageProperties bean.

The only property that we've configured within MessageProperties is the greeting property. So **messages.greeting** will be injected into the greeting variable of this class. The default

Spring Boot Microservices: Getting Started

greeting that we've configured here on line 13 is Good morning. So if no application property is specified for messages.greeting, the default greeting will be *Good morning*.

```
private String greeting = "Good Morning";
```

We have to set up getters and setters for this greeting property as well within this class.

Let's head over to the GreetingController where we've updated our code to get the greeting from this MessageProperties class.

```
GreetingController.java
1 package com.springboot.springinitializr.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5
6 @Controller
7 public class GreetingController {
8
9     @Autowired
10    private MessageProperties properties;
11
12
13    @GetMapping("/greeting")
14    public String greeting(@RequestParam String name, Model model) {
15        String message = properties.getGreeting() + name;
16        model.addAttribute("greetingMessage", message);
17        return "homePage";
18    }
19 }
```

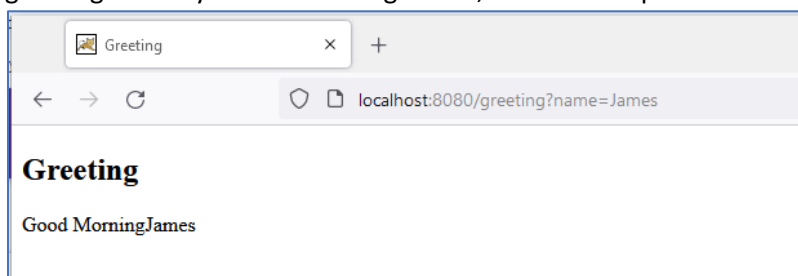
The way we get a reference to that object is via Spring's dependency injection.

The **@Autowired** annotation on MessageProperties basically tells spring to inject this bean into our greeting controller.

There is another change to our code on line 18. We get the greeting that we are going to render to the user using properties.getGreeting, we append the name as well.

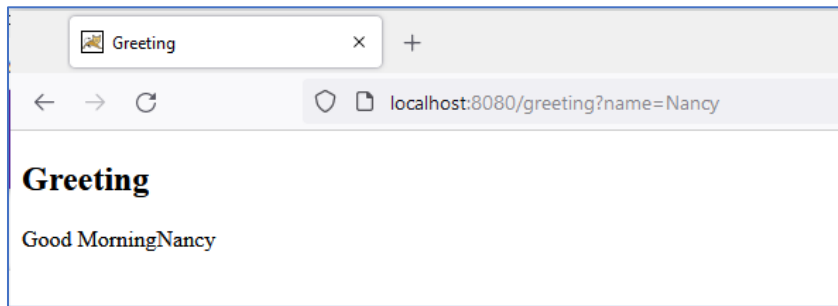
Now we render the homPage as before by passing on the greetingMessage.

- Let's run this code and switch over to localhost, greeting?name=James. And notice that the greeting now says Good morning James, the default specified in our MessageProperties file.



Spring Boot Microservices: Getting Started

Let's change the value of our request parameter to Nancy. Name is equal to Nancy and the greeting says Good morning Nancy.



Back to our Eclipse editor. This time we'll specify the greeting that we want displayed to the user using the application.properties file.

Now in order for this property set up to work correctly, in order to make our Eclipse editor aware of this property, we'll need to add another dependency to our pom file.

pom.xml

```
36<dependency>
37  <groupId>org.springframework.boot</groupId>
38  <artifactId>spring-boot-configuration-processor</artifactId>
39</dependency>
```

The **spring-boot-configuration-processor** specified on line 38 ensures that our application is aware of the metadata associated with the property. This property will be recognized when we add it to application properties.

So head over to **application properties**, messages.greeting is equal to Boo!. The editor is not aware of the property yet. Select the property, and select the option create metadata for messages.greeting.

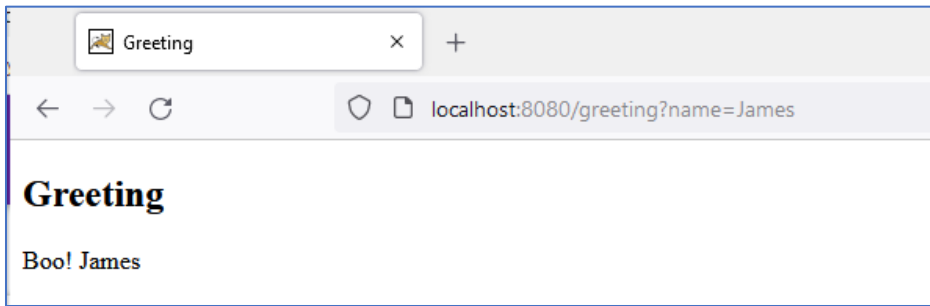
application.properties

```
application.properties
1 messages.greeting= Boo! |
```

Let's now run our application and see whether the greeting from this **application.properties** file is picked up by our app. Always do Maven Update Project after updating pom.xml, and maven build before running the application.

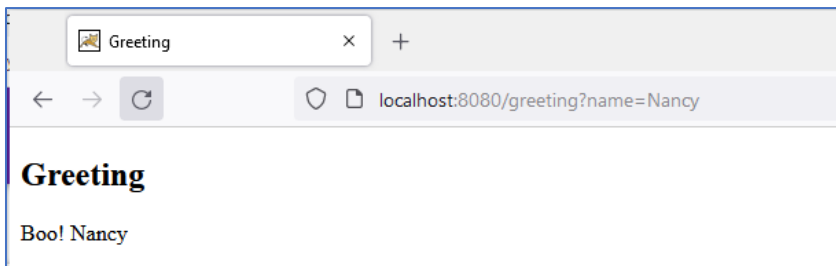
Spring Boot Microservices: Getting Started

Go to the URL localhost:8080, greeting name equal to James.



And you can see here that James is greeted using Boo, the message specified in our application.properties.

Let's greet Nancy now, and Nancy is greeted using Boo as well.



We've successfully specified user-defined properties within our application.properties configuration file.

Spring Boot Microservices: Getting Started

Summary

In this course, we introduced the Spring Boot framework, which is an extension of the Spring framework used to work with Spring modules for data access, building front ends, and everything else that we need in an enterprise-grade application.



We saw how Spring Boot abstracts away the complexities of dealing with Spring modules and versions. We started off by seeing how we can get our first Spring Boot application up and running using Apache Maven. We explored the use of starter templates for dependency management for common Spring applications, and the use of the parent pom which gave us sensible defaults for our Spring dependencies. We then ran a very simple web server using Spring Boot. We saw how Spring Boot applications just work when our web application automatically ran on an embedded Tomcat server. We saw how we could change the default server to be a Jetty server using dependency exclusions. We also configured our application to be packaged as a WAR file to be deployed on an external server.

We also explored configuration management using the `application.properties` file in our Spring Boot app. And the internationalization of our applications messages using `message.properties`.

Spring Boot Microservices: Getting Started

Quiz

1. What is Spring?
A framework that makes it easy for Java developers to manage dependencies
A design pattern that allows for dependency injection
A framework which is meant to build user interfaces
✓ A framework which provides a convenient way for different components to be integrated
2. Which of these statements about Spring Boot is true?
✓ It is an extension of the Spring framework
It replaces the original Spring model of development
✓ It makes developing in Spring easier
It is the latest version of the Spring framework
3. What is Apache Maven?
An IDE for writing Java code
A compiler for Java code which simulates the JVM runtime
A quick text editor for configuration files
✓ A project and build management tool for Java and other projects
4. What do you need to configure so you can run mvn commands from your terminal prompt?
Your system settings
✓ \$PATH environment variable
Your Eclipse IDE
\$JAVA_HOME environment variable
5. What is a Maven archetype?
The object model of our application
A build management tool for Java projects
✓ A project template which contains the basic structure of our Maven project
An IDE for Java projects
6. Which is the one annotation that you would apply to the class which contains the main() method for your Spring Boot app?
@SpringBootConfiguration
✓ @SpringBootApplication
@ComponentScan
@EnableAutoConfiguration

Spring Boot Microservices: Getting Started

7. What is the starter dependency descriptor you should use in order to use the embedded Tomcat server in SpringBoot?
 - ✓ **spring-boot-starter-web**
 - spring-boot-starter-jetty*
 - spring-boot-starter-tomcat*
 - spring-boot-starter*
8. Which are the embedded servers that the spring-boot-starter-web dependency supports?
 - httpserver*
 - ✓ **Jetty**
 - ✓ **Undertow**
 - IIS*
 - ✓ **Tomcat**
9. When deploying your web application as an external war file what are the first two details you need to specify in your app?
 - ✓ **Set the Maven packaging as "war"**
 - Have your configuration have the annotation @ExternalWar*
 - ✓ **Have your configuration derive from SpringBootServletInitializer**
 - Set the Maven packaging as external-war*
10. What is the Spring Initializr?
 - A dependency management system explicitly for Spring modules*
 - ✓ **A web interface used to specify the project structure and dependencies for a Spring Boot project**
 - A new technique to inject Spring beans which make up the dependencies of our project*
 - A no-code deployment server which is commonly used for Spring web applications*
11. What is Thymeleaf in Spring Boot?
 - The internationalization engine which generates messages in different languages*
 - The model engine which processes data in our web application*
 - ✓ **The template engine which transforms and displays views in a web application**
 - The dependency engine responsible for injecting beans in a Spring application*
12. What is the Spring Tools Suite?
 - A suite of tools which makes deployment of Spring applications easier*
 - ✓ **A suite of tools integrated with Eclipse to make development in Spring easier**
 - A suite of tools which makes dependency management in Spring easier*
 - A suite of tools integrated with Eclipse which makes error reporting in Spring easier*

Spring Boot Microservices: Getting Started

13. What is the property in Spring which helps you change the port your application is running on?

✓ `server.port`

`port`

`application.port`

`server.address.port`

14. What is the annotation that tells Spring that we have external properties specified in a Java file?

`@Bean`

`@Properties`

✓ `@ConfigurationProperties`

`@ExternalProperties`