



## What We're Going to Cover

---

In this series of tutorials and demos, you're going to learn about:

- Spring
- Spring Boot
- How to create a Spring Boot Hello World application.
- How to create a Spring Boot web application.
- How to create a Spring Boot RESTful service.
- And how to have fun doing it!

You'll accomplish all this by creating a "Hello World" Spring Boot application utilizing Spring Initializr – always a good place to start. You'll then extend your knowledge of Spring by applying some useful patterns that are commonly used with web applications. We'll round that off by further expanding your web application into a RESTful web service that you'll be able to access via web browser.

Let's get started...

## Table of Contents

<b><i>What We're Going to Cover</i></b>	<b><i>i</i></b>
<b><i>Frameworks</i></b>	<b><i>1</i></b>
<b>What is a Framework?</b>	<b>1</b>
<b>Features</b>	<b>1</b>
Inversion of Control	1
Extensibility	1
Non-Modifiable Framework Code	1
<b><i>What is Spring?</i></b>	<b><i>2</i></b>
<b>History</b>	<b>2</b>
<b>Spring Projects</b>	<b>3</b>
<b><i>What is Spring Boot?</i></b>	<b><i>5</i></b>
<b>Notable Features</b>	<b>5</b>
<b>Intelligent Auto-Configuration</b>	<b>5</b>
<b>Being Stand-Alone</b>	<b>5</b>
Starting a Java-Based Web Application	5
Starting a Spring Boot Application	5
<b>Opinionated</b>	<b>6</b>
<b><i>Spring Boot Hello World</i></b>	<b><i>8</i></b>
<b>Create the Application</b>	<b>8</b>
<b>Project Setup</b>	<b>9</b>
<b>Running a Build</b>	<b>12</b>
<b>The Maven Build File</b>	<b>14</b>
<b>Project Structure</b>	<b>17</b>
<b>Running Hello World</b>	<b>17</b>
<b>The Code</b>	<b>18</b>
<b>Running the Application</b>	<b>19</b>
<b>Check Application Status</b>	<b>20</b>
<b>Let's Review</b>	<b>20</b>
<b>Knowledge Check</b>	<b>20</b>
<b><i>Expanding on a Good Thing – Adding Some Patterns</i></b>	<b><i>21</i></b>
<b>Why Use Patterns?</b>	<b>21</b>
<b>MVC</b>	<b>21</b>
<b>3-Tier</b>	<b>22</b>
<b>Merging Patterns</b>	<b>23</b>
<b>Applying the Patterns</b>	<b>24</b>

HelloMessage	25
HelloRepository	26
HelloService	27
HelloController	28
HelloApplication	29
application.properties	30
<b>Let's Review</b>	<b>34</b>
<b>Knowledge Check</b>	<b>34</b>
<b><i>Expanding on a Good Thing – Spring REST Services</i></b>	<b>35</b>
HelloController	35
<b>Let's Review</b>	<b>38</b>
<b>Knowledge Check</b>	<b>38</b>
<b><i>Wrapping it All Up</i></b>	<b>39</b>
<b><i>Additional Resources</i></b>	<b>39</b>
<b><i>Knowledge Check Answers</i></b>	<b>41</b>
Check 1	41
Check 2	41
Check 3	42

### What is a Framework?

---

In computer programming, a *software framework* is an abstraction in which software, providing generic functionality, can be selectively changed by additional user-written code, thus providing application-specific software. It provides a standard way to build and deploy applications and it is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions.

Software frameworks may include support programs, compilers, code libraries, tool sets, and application programming interfaces (APIs) that bring together all the different components to enable development of a project or system.

### Features

---

Frameworks have key distinguishing features that separate them from normal libraries:

#### **Inversion of Control**

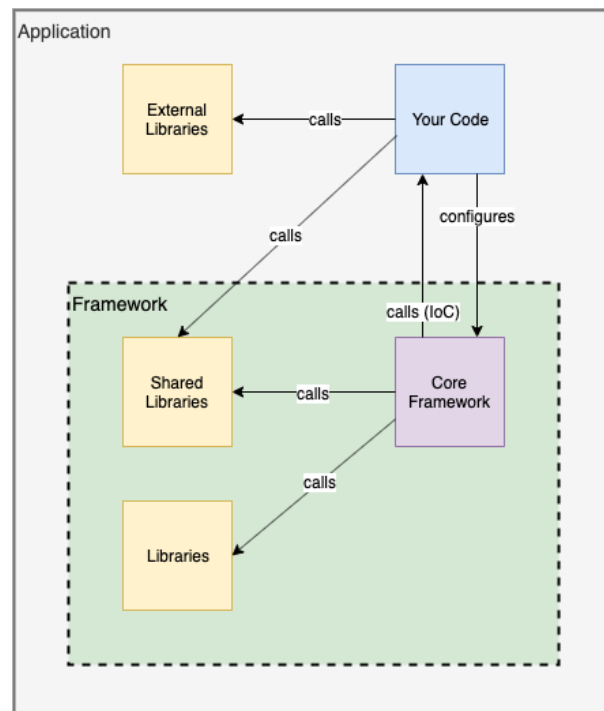
In a framework, unlike in libraries or in standard user applications, the overall program's flow of control is not dictated by the caller, but by the framework.

#### **Extensibility**

A user can extend the framework – usually by selective overriding – or programmers can add specialized user code to provide specific functionality.

#### **Non-Modifiable Framework Code**

The framework code, in general, is not supposed to be modified, while accepting user-implemented extensions. In other words, users can extend the framework, but cannot modify its code.



Source: [https://en.wikipedia.org/wiki/Software\\_framework](https://en.wikipedia.org/wiki/Software_framework)

The term Spring means different things in different contexts. It can be used to refer to the Spring Framework project itself, which is where it all started. Over time, other Spring projects have been built on top of the Spring Framework. Most often, when people say Spring, they mean the entire family of projects.

The Spring Framework is divided into modules. Applications can choose which modules they need for their specific use case. At the heart of Spring are the modules of the core container, including a configuration model and a *dependency injection* mechanism. Beyond that, the Spring Framework provides foundational support for different application architectures, including *messaging*, *transactional data* and *persistence*, and *web*.

## History

Spring came into being in 2003 as a response to the complexity of the early J2EE specifications. While some consider [Java EE](#) and Spring to be in competition, Spring is, in fact, complementary to Java EE. The Spring programming model does not embrace the Java EE platform specification; rather, it integrates with carefully selected individual specifications from the EE umbrella. These include (but are not limited to):

- Servlet API ([JSR 340](#))
- WebSocket API ([JSR 356](#))
- Concurrency Utilities ([JSR 236](#))
- JSON Binding API ([JSR 367](#))
- Bean Validation ([JSR 303](#))
- JPA ([JSR 338](#))
- JMS ([JSR 914](#))

The Spring Framework also supports the Dependency Injection ([JSR 330](#)) and Common Annotations ([JSR 250](#)) specifications, which application developers may choose to use instead of the Spring-specific mechanisms provided by the Spring Framework.

## Spring Projects

---

In the beginning, the Spring Framework started as a Dependency-Injection container and grew into a full, rich and mature ecosystem. Over the years, Spring has produced many notable sub projects, including:

- Spring Framework (Core)
- Spring Boot
- Spring Cloud
- Spring Data
- Spring Cloud Data Flow
- Spring Integration
- Spring Batch
- Spring Security
- Spring AMQP (Messaging)
- Spring Mobile
- Spring Web Flow
- Spring Web Services
- Spring State Machine

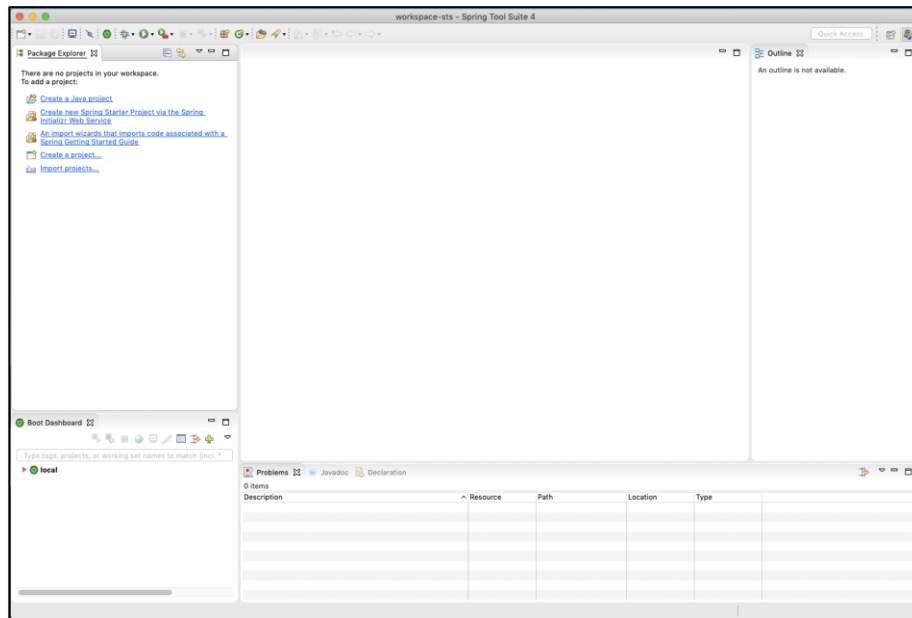
Even with all of these projects, one of Spring's main benefits is still it's dependency injection capabilities.



<https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>

Dependency Injection

Spring has even evolved its own IDE, based on eclipse, called Spring Tools 4 (or STS). Spring Tools 4 is free and you can check out and download a copy from their site at <https://spring.io/tools>



Source: <https://spring.io/>

Source: <https://spring.io/tools>

Source: <https://www.baeldung.com/spring-intro>

Source: <https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html>



## XML → Annotations

Just as a side note, as you learn and progress through Spring, you'll notice a lot of code examples that involve XML configuration files. XML is the original way of configuring Spring and has been replaced with Java annotations in the newer versions of Spring (since Spring 2.5). Anything you can do with XML, you can do with annotations, which is the preferred method now.



<https://springframework.guru/spring-framework-annotations/>

Spring Annotations



Normally, when using the Spring framework, you have to manually configure nearly every aspect of your application. Spring Boot removes this hurdle for you by providing opinionated configurations that work right out of the box to get your Spring application up and running in just seconds.

### Notable Features

---

- Auto-Configuration
- Stand-Alone
- Opinionated

### Intelligent Auto-Configuration

---

The intelligent auto-configuration attempts to auto-configure your application based on what dependencies you added to your build file. It's contextually aware and smart.

### Being Stand-Alone

---

You may think that running a Java application is easy. After all, in eclipse, all you do is right-click on your driver class and choose Run. To be honest, it's not that simple.

#### Starting a Java-Based Web Application

---

- First thing you do is compile and package your application.
- Next, you have to choose which type of web server you want to use and download and install it.
- You next have to configure the web server.
- After that, you have to organize the deployment process.
- And lastly, you start your web server.

#### Starting a Spring Boot Application

---

- Compile and package your application.
- Run it with a simple java command or click on run configuration run button in your IDE.

Actually, most IDEs will execute the previous two steps with one click, once a run configuration is created within the IDE. This makes development much quicker.



## Opinionated

---

This is probably one of the greatest benefits of using Spring Boot – it's highly opinionated configurations. When you write Java applications, you have tons of choices, starting with:

- The Web Framework
  - Tomcat
  - GlassFish
  - Jetty
  - Apache TomEE
  - Oracle WebLogic
  - WebSphere
  - WildFly
  - Apache Geronimo
  - JBoss
- The Logging Framework
  - Java native (java.util.logging)
  - SLF4J (Simple Logging Facade for Java)
  - Log4J (Logging for Java)
- The Collection Framework
  - Java Collections
  - Google Guava
  - Jakarta Commons Collections
- UI Framework
  - ReactJS (Facebook)
  - Angular (Google)
  - Vue
  - Thymeleaf
  - Bootstrap (Twitter)
  - JQuery
- Database
  - Oracle
  - MySQL
  - MariaDB
  - PostgreSQL
  - Cassandra
  - MongoDB

- Messaging Service
  - RabbitMQ
  - Kafka
- Build Tool
  - Ant
  - Maven
  - Gradle

Despite this, in the most cases, the developers use the same most popular libraries. All that Spring Boot does is load and configure them in the most standard configuration. Hence, the developers don't need to spend a lot of time configuring the same thing over and over again, and have more time for writing code and satisfying business needs.

Source: <https://dzone.com/articles/what-is-spring-boot>

Source: <https://spring.io/projects/spring-boot>

Source: <https://www.codeinwp.com/blog/angular-vs-vue-vs-react/>





Every good application starts at the beginning. This usually means some type of *Hello World*. Spring & Spring Boot are no different. So, let's start by creating a basic Hello World application using Spring Boot.

We're going to accomplish this by utilizing a really fun and easy website called [Spring Initializr](#). This site is a web application that can generate a Spring Boot project – structure and all – for you from scratch. All you have to do is provide it some basic information such as which version of Java you want to use, which build tool you want to use and some basic project metadata.

Just remember, Spring Boot applications can be created in several ways – this is just probably the easiest.

### Create the Application

---

1. In your browser, navigate to <https://start.spring.io/>
2. Next to **Project**, select *Maven Project*.
3. Next to **Language**, select *Java*.
4. Next to **Spring Boot**, select the latest version (usually far right menu pick). 2.1.8 at this writing.
5. Next to Project Metadata:
6. Enter *com.example* under **Group**
7. Enter *spring-boot-hello-world* under **Artifact**
8. Expand **Options** node.
9. Enter *Spring Boot Demo* under **Name**.
10. Enter *Demo project for Spring Boot* under **Description**.
11. Enter *com.example.springboot* under **Package Name**.
12. Select *Jar* under **Packaging**.
13. Select *8* under **Java**.
14. Next to **Dependencies**, click on the menu icon.
15. Under **Developer Tools**, select *Spring Boot DevTools*.
16. Under **Web**, select *Spring Web*.
17. Under **SQL**, select *Spring Data JPA and H2 Database*.
18. Under **Ops**, select *Spring Boot Actuator*.

19. At the bottom of the page, click **Generate** to download a zip file of your project.

## Project Setup

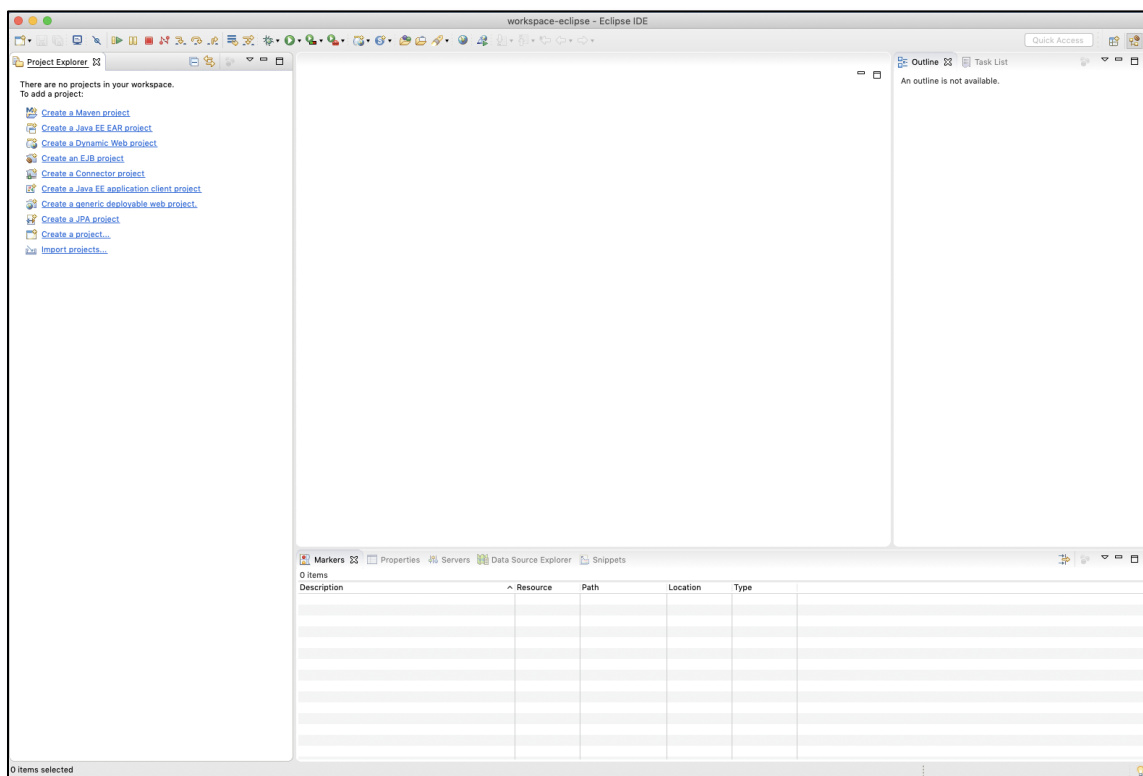
---

20. Once your project is downloaded you need to move it to a location of your choosing (generally a workspace folder if you're using eclipse).

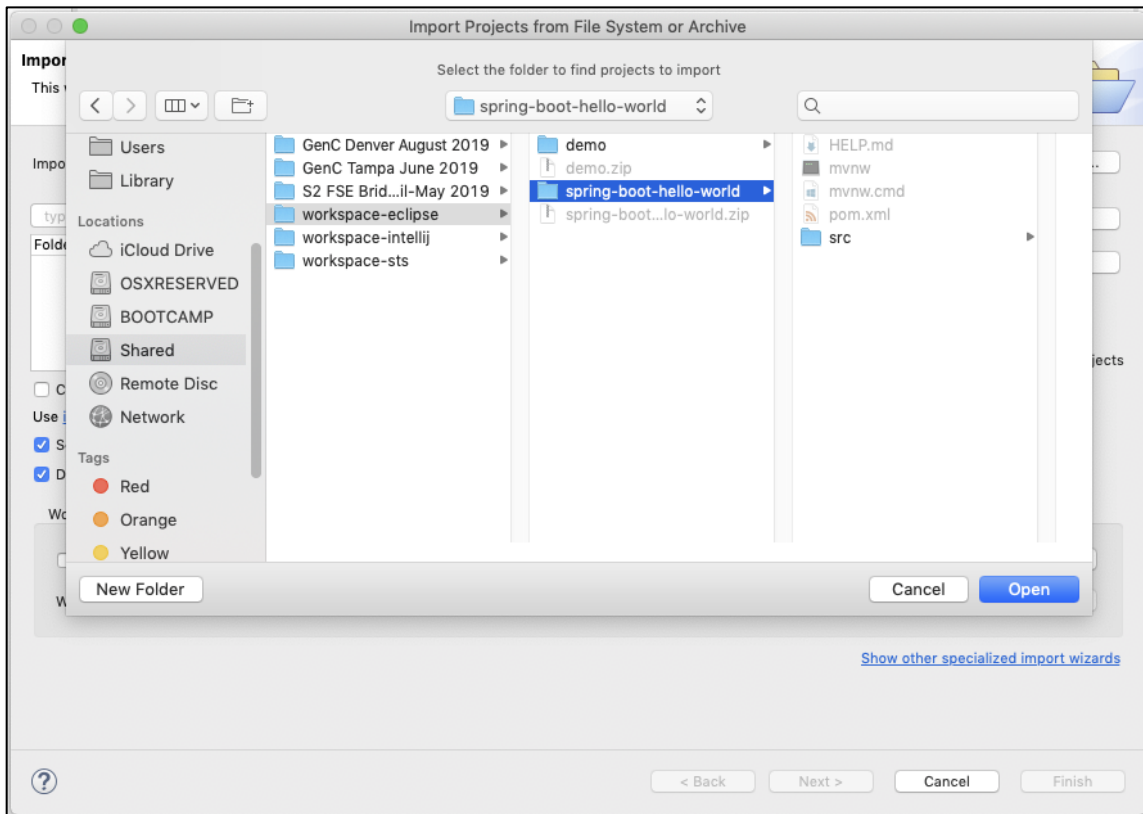
21. Once the zip file is moved, extract it into the workspace folder.

22. Open eclipse and specify the folder you unzipped your project into as your workspace.

23. Close the welcome tab (if open) and you should see something similar to below.

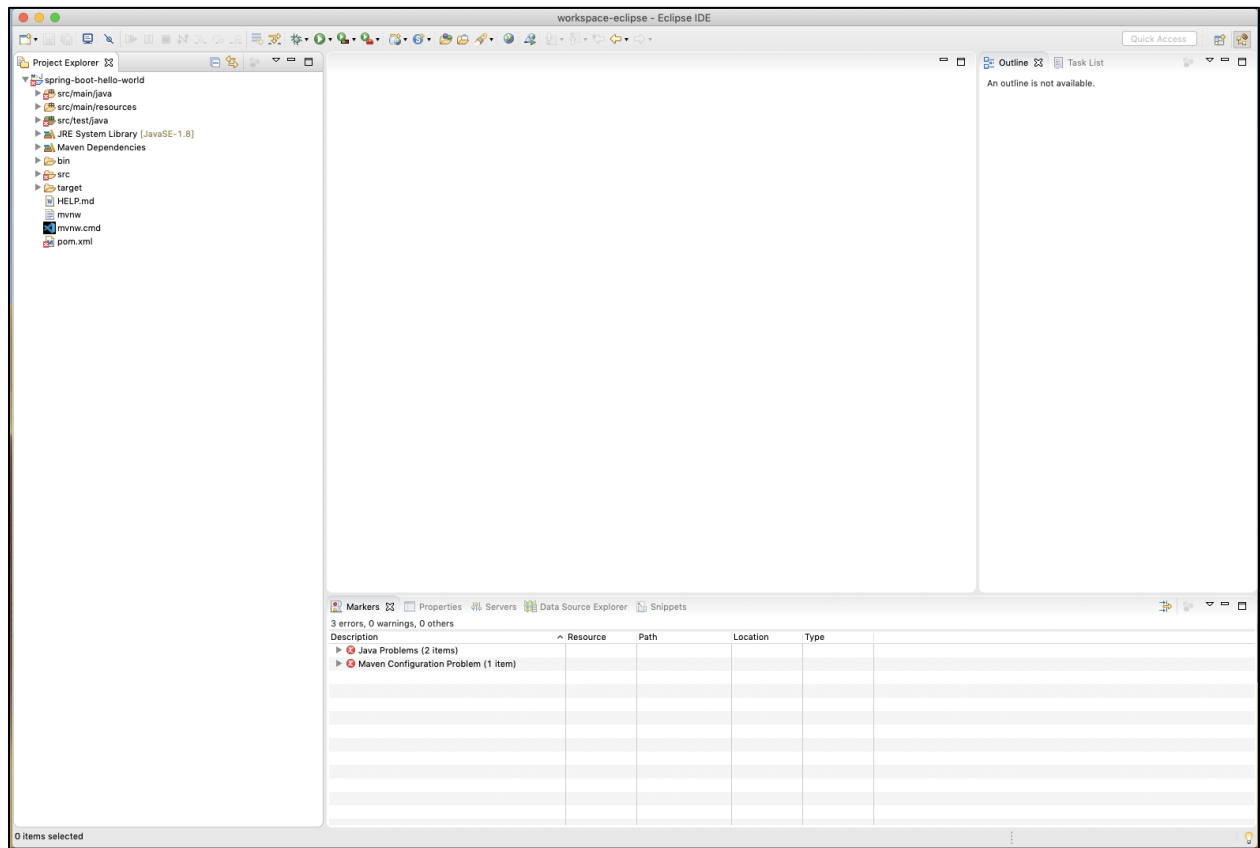


24. Next, we need to add our project to eclipse. We do this by selecting File → Open Projects from File System.
25. In the wizard that pops up, under **Import Source**, select Directory, then choose the directory that contains your project. It will be the one you unzipped into your workspace.



26. Select **Open**.
27. Select **Finish**.

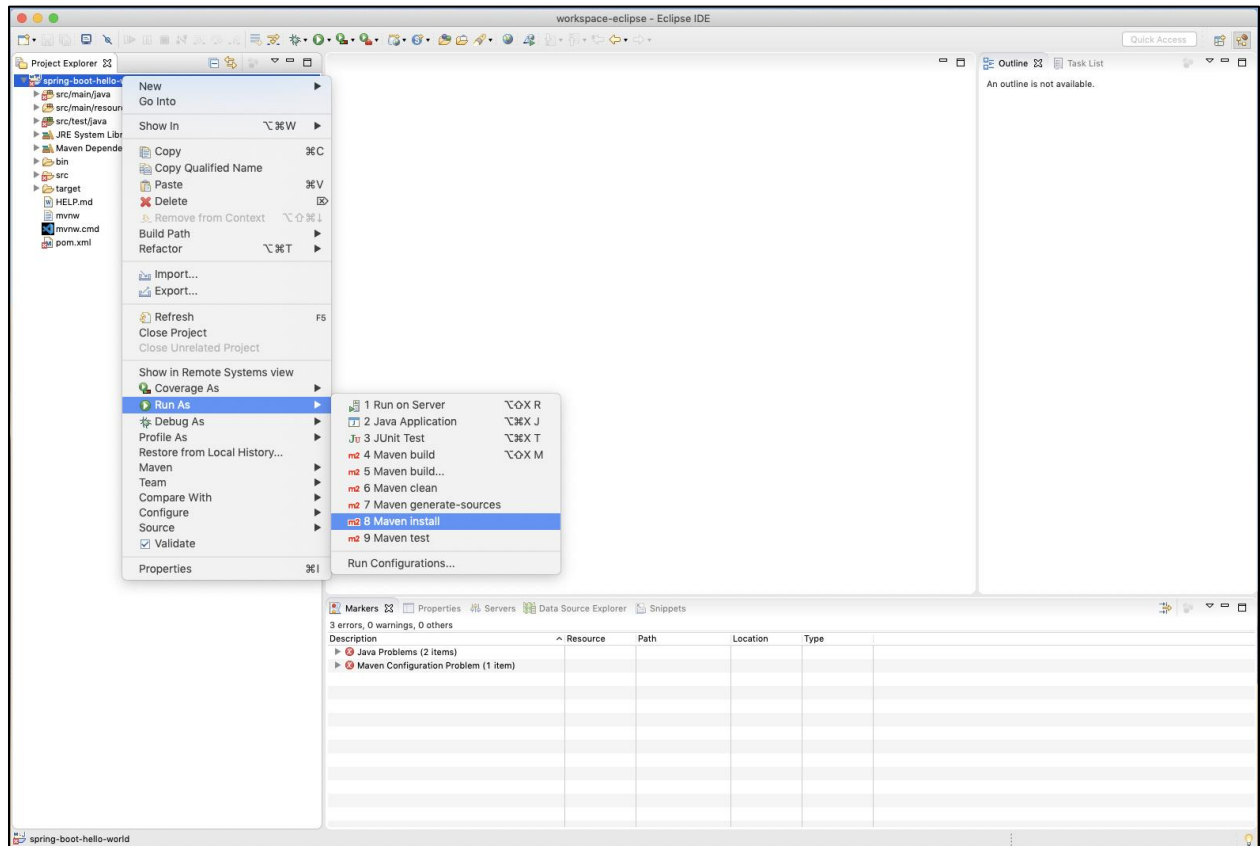
28. You should now see the project in your Project Explorer.



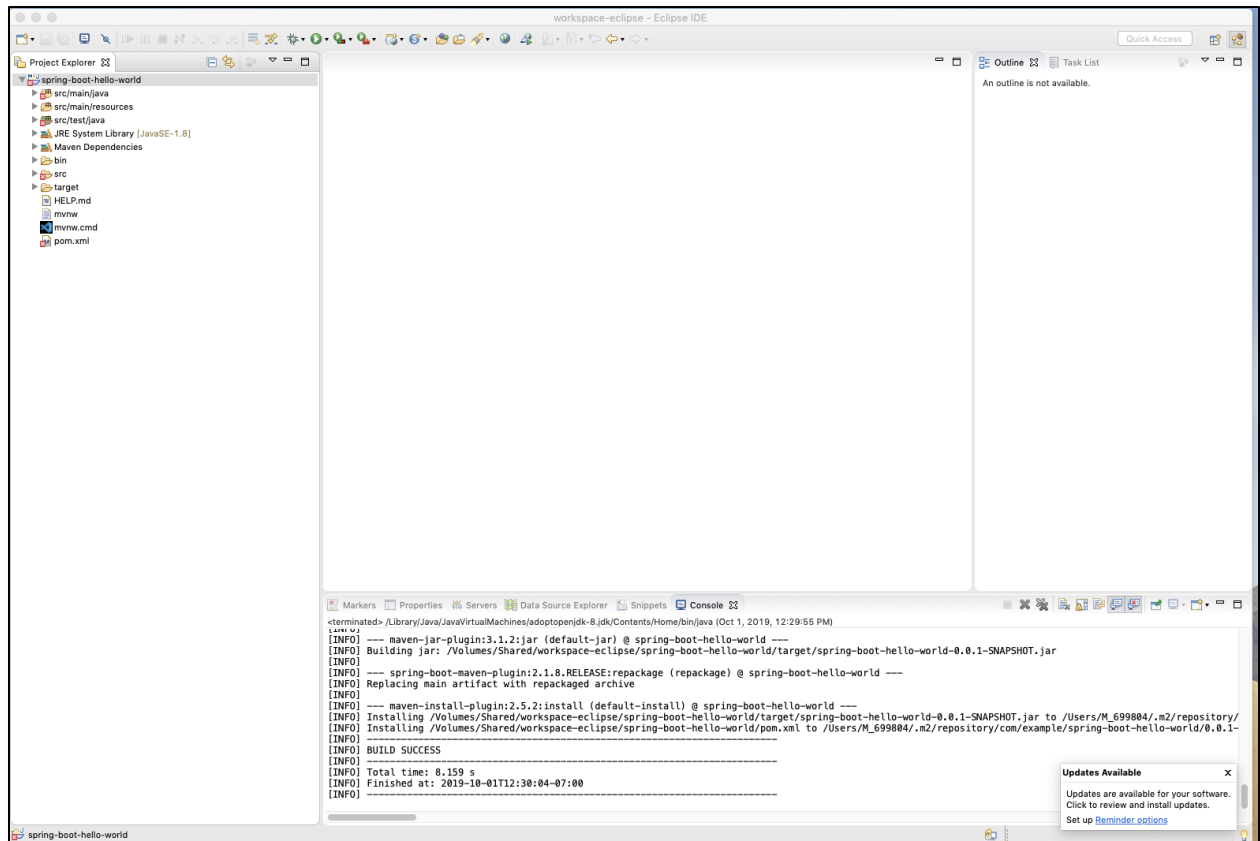
29. Notice there are some errors showing - those are normal at this point. We haven't run our build yet.

## Running a Build

30. Right-click on the *parent node* of your project in the Project Explorer.
31. Mouse over **Run As**.
32. Click on **Maven Install**. This will run your Maven build and download all of the Spring dependencies we need for our project.



33. Once the Maven build runs, you should see a message in the Console tab that says BUILD SUCCESS. If you see that, you're good.
34. If you get a BUILD FAILURE, you have to scan through the Console logs to determine what the problem is. Once fixed, go back and repeat the previous steps until you get a BUILD SUCCESS message.
35. Typical problems could be anything from compile issues to Maven not being able to download dependencies from Maven Central. Read the logs carefully to determine the exact problem.





## The Maven Build File

---

Let's take a look at the build file. Open the file called *pom.xml* – it's in the root of your project. This file is written in a language called XML (Extensible Markup Language). XML is merely a language used for transporting data. It does nothing in and of itself – only *marks* data so it can be interpreted by a program.

For more information on XML, check out the W3Schools primer on it at [https://www.w3schools.com/xml/xml\\_what.asp](https://www.w3schools.com/xml/xml_what.asp)

Let's go over the pieces of the build file...

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

This piece of code is the header of our build file and very rarely, if ever, changes. You'll always see this at the top of each of your Maven build files.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.8.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

This parent element tells Maven that this build is a child of the spring-boot-starter-parent build. This is how we make this build a Spring Boot build and not just a regular Maven build. In this case, we're using Spring Boot version 2.1.8.RELEASE, which is the most current as of this writing.

```
<groupId>com.example</groupId>
<artifactId>spring-boot-hello-world</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>spring-boot-hello-world</name>
<description>Demo project for Spring Boot</description>
```

These elements are project metadata and were created from some of the values we provided to Spring Initializr. These values can be changed and will only affect how the project artifact is referenced. In other words, we can change the version, groupId, artifactId, name and description to suite the project we're on.

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

This properties element is where we can specify properties that are used throughout the build. The only one we're using right now is the java.version. If we were to change our Java version to 11, we would need to change 1.8 to 11 inside the java.version XML tags.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

The dependencies element is where we specify each individual dependency to our project. We can add dependencies as needed here.

You may be asking yourself, “Where, oh where, do I get all that code inside those dependency elements for a given dependency?” Glad you asked! It's actually easy.

For any given library, you can almost always find it on **Maven Central**. The easiest way is to Google the library name you want along with the words Maven Central and the first search return will be a link to that library on Maven Central.



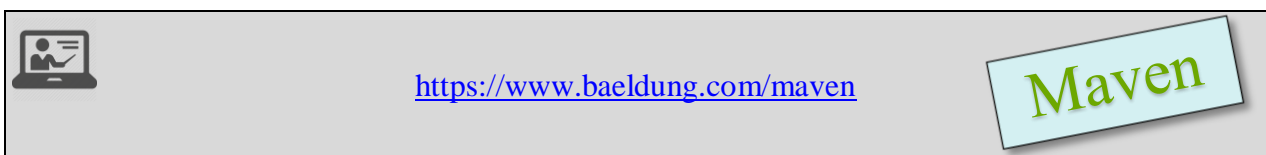
For example, if I wanted to add the Guava library, I would Google “Guava Maven Central”. This will return a link for the Guava library on Maven Central. From there, I can then pick the version I want – say 28.1-jre – which will switch to a page for that version. Once there, I can select the Maven tab just below the header and the GAVC coordinates (pronounced GAV - they are called that from **Group**Id, **Artifact**Id, **V**ersion, **C**lassifier) are displayed there in the box below to be copied into your build file.

Maven Central is a huge library repository that, at the time of this writing, has 15.3

million artifacts (libraries) in it.

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

The build element has a lot of purposes. In our case, we’re using it to add the spring-boot-maven-plugin which tells Maven how to package our Spring Boot JAR file when we run a Maven Build.

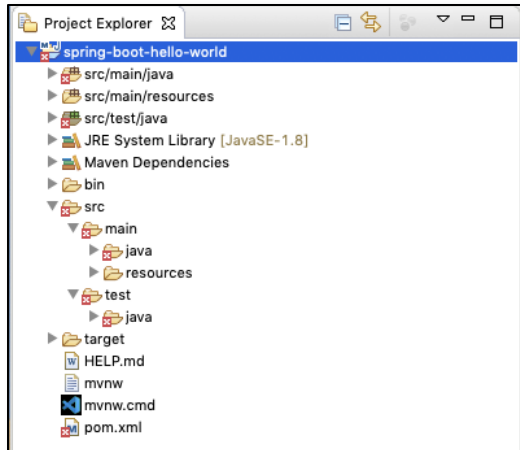


## Project Structure

---

Let's take a look at the project structure now.

In your Project Explorer, expand the *src* node (*src* is just short for source).



You'll notice there are two folders under *src* – *main* and *test*. The *main* folder is where all of your source code lives including resources. The *test* folder is where all of your unit and integration tests live, including all testing resources.

Resources can include things like *application.properties* files, static files as well as template files. The last two are used with UI components of a full web application while the *application.properties* has key-value pairs in them that can overwrite or set properties within Spring and other frameworks such as Spring Data JPA or MySQL.

The main thing to note about this standard Maven structure is that your Java source code is now located under *src/main/java*, while your test code is located under *src/test/java*.

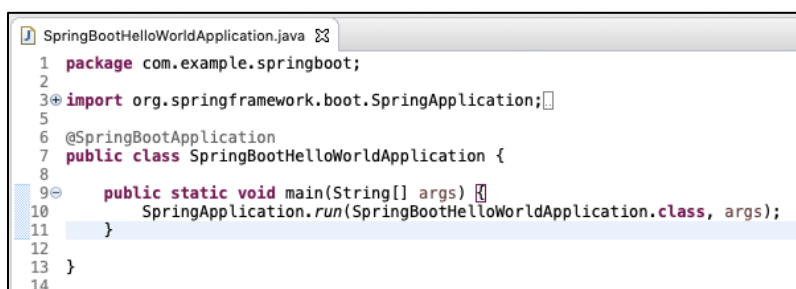
## Running Hello World

---

First, we need to add the standard greeting for our Hello World, because it doesn't come with it when the code is auto-generated.

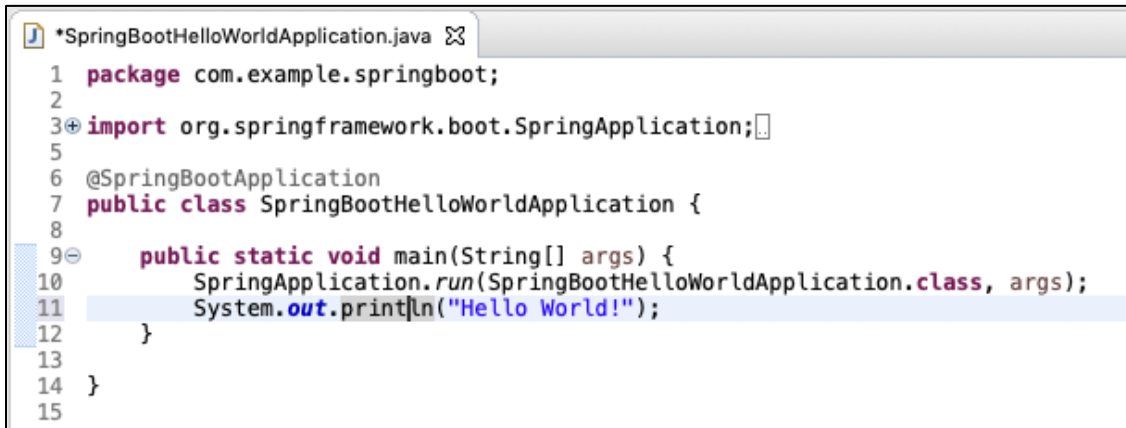
36. Navigate to and open the driver class inside your source folder (*SpringBootHelloWorldApplication.java*). In our case, it's located in *src/main/java/com/example/springboot*.

37. When you open the file, you should see something like this:



```
1 package com.example.springboot;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class SpringBootHelloWorldApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringBootHelloWorldApplication.class, args);
11     }
12 }
13
14
```

38. Go ahead and add the standard Hello World greeting below line 10:



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

39. Save your file.

## The Code

---

Looking at the code above, the annotation on line 6 does a lot for us.

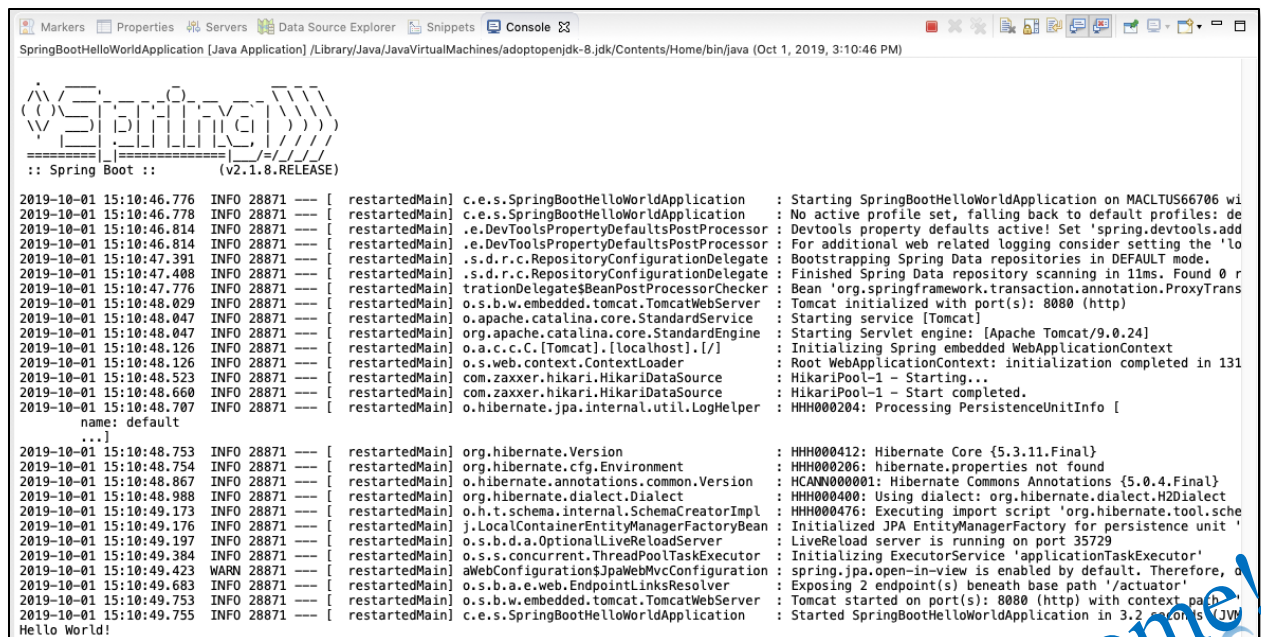
The first thing it does is mark the driver class as a Spring Boot application. It also has the advantage of encapsulating several other annotations, including `@Configuration`, `@EnableAutoConfiguration` and `@ComponentScan`.

- `@Configuration` marks the class as a configuration class. These classes are handled in specific ways by Spring.
- `@EnableAutoConfiguration`, as the name implies, enables auto configuration on the application. It means Spring Boot looks for auto-configuration beans on its classpath and automatically applies them. Beans are just special classes in your code that are managed by Spring.
- `@ComponentScan` allows Spring to look for specially marked classes and add them to its IOC container to be managed by Spring. This allows us to use dependency injection on these marked classes. (Examples of some of the annotations used to mark classes as beans are `@Controller`, `@Service` and `@Repository`)

## Running the Application

Now we're ready to run the application.

40. **Right-click** anywhere on the code, then select **Run-As**, then select **Java Application**.
41. If you get a warning about there being errors, just click **Proceed**.
42. The app will start and you'll get the standard Spring Boot header followed by a couple dozen logging lines. At the bottom of the lines (or near the bottom), you'll see your Hello World message.



```
Markers Properties Servers Data Source Explorer Snippets Console
SpringBootHelloWorldApplication [Java Application] /Library/Java/JavaVirtualMachines/adoptopenjdk-8-jdk/Contents/Home/bin/java (Oct 1, 2019, 3:10:46 PM)

:: Spring Boot ::
(v2.1.8.RELEASE)

2019-10-01 15:10:46.776 INFO 28871 --- [ restartedMain] c.e.s.SpringBootHelloWorldApplication : Starting SpringBootHelloWorldApplication on MACLTUS66706 wi
2019-10-01 15:10:46.778 INFO 28871 --- [ restartedMain] c.e.s.SpringBootHelloWorldApplication : No active profile set, falling back to default profiles: de
2019-10-01 15:10:46.814 INFO 28871 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add
2019-10-01 15:10:46.814 INFO 28871 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'lo
2019-10-01 15:10:47.391 INFO 28871 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data repositories in DEFAULT mode.
2019-10-01 15:10:47.408 INFO 28871 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 11ms. Found 0 r
2019-10-01 15:10:47.776 INFO 28871 --- [ restartedMain] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTrans
2019-10-01 15:10:48.029 INFO 28871 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-10-01 15:10:48.047 INFO 28871 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-10-01 15:10:48.047 INFO 28871 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.24]
2019-10-01 15:10:48.126 INFO 28871 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-10-01 15:10:48.126 INFO 28871 --- [ restartedMain] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 131
2019-10-01 15:10:48.523 INFO 28871 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2019-10-01 15:10:48.660 INFO 28871 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2019-10-01 15:10:48.707 INFO 28871 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
name: default
...]
2019-10-01 15:10:48.753 INFO 28871 --- [ restartedMain] org.hibernate.Version : HHH000412: Hibernate Core {5.3.11.Final}
2019-10-01 15:10:48.754 INFO 28871 --- [ restartedMain] org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
2019-10-01 15:10:48.867 INFO 28871 --- [ restartedMain] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.0.4.Final}
2019-10-01 15:10:48.988 INFO 28871 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
2019-10-01 15:10:49.173 INFO 28871 --- [ restartedMain] o.h.t.schema.internal.SchemaCreatorImpl : HHH000476: Executing import script 'org.hibernate.tool.sche
2019-10-01 15:10:49.176 INFO 28871 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit '
2019-10-01 15:10:49.197 INFO 28871 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2019-10-01 15:10:49.384 INFO 28871 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-10-01 15:10:49.423 WARN 28871 --- [ restartedMain] aWebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, o
2019-10-01 15:10:49.683 INFO 28871 --- [ restartedMain] o.s.b.a.e.web.EndpointLinksResolver : Exposing 2 endpoint(s) beneath base path '/actuator'
2019-10-01 15:10:49.753 INFO 28871 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path
2019-10-01 15:10:49.755 INFO 28871 --- [ restartedMain] c.e.s.SpringBootHelloWorldApplication : Started SpringBootHelloWorldApplication in 3.2 seconds [JVM
Hello World!
```

And there you have it! Your first Spring Boot application!

Awesome!

## Check Application Status

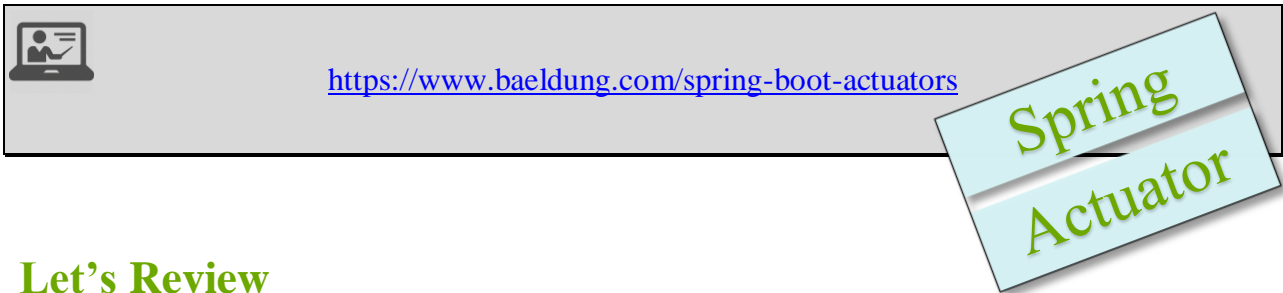
---

Because we are using **Spring Actuator**, we can check the status of our application through a web browser.

43. Do this by navigating to <http://localhost:8080/actuator/health>

44. You should see the actuator message `{"status": "UP"}`

This functionality is provided by Spring Actuator and can be customized through the use of custom endpoints.



## Let's Review

---

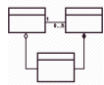
In these first sections, you learned about:

- Spring
  - IOC Container.
  - Dependency Injection.
- Spring Boot
  - Auto-Configuration
  - Stand-Alone
  - Opinionated
- Spring Initializr
- Maven Build File Components
- Running a Spring Boot Application
- Simple Application Monitoring

## Knowledge Check

---

- What Spring framework allows you to monitor your application?
- What is the biggest benefit to using Spring?
- What are the 3 annotations that are encapsulated by the `@SpringBootApplication` annotation?



### Why Use Patterns?

Software professionals may be familiar with the term *Design Patterns*, but many have no idea of where they come from and what they truly are. Consequently, some do not see the value and benefits design patterns bring to the software development process, especially in the areas of maintenance and code reuse.

Design patterns are commonly defined as time-tested solutions to recurring design problems. The term refers to both the description of a solution that you can read, and an instance of that solution as used to solve a particular problem.

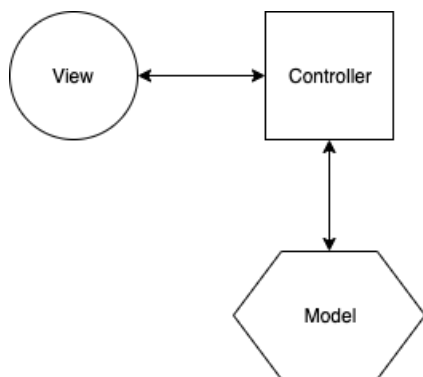
Design patterns have two major benefits. First, they provide you with a way to solve issues related to software development using a proven solution. The solution facilitates the development of highly cohesive modules with minimal coupling. They isolate the variability that may exist in the system requirements, making the overall system easier to understand and maintain. Second, design patterns make communication between designers more efficient. Software professionals can immediately picture the high-level design in their heads when they refer the name of the pattern used to solve a particular issue when discussing system design.

Other major benefits include code reuse and better maintainability, both desirable attributes of any good software solution.

Source: <https://www.developer.com/design/article.php/1474561/What-Are-Design-Patterns-and-Do-I-Need-Them.htm>

### MVC

Probably *the* most common pattern used in web applications is one called **MVC**. This stands for **Model – View – Controller**.



The **View(s)** typically consists of various HTML, JavaScript, CSS, documents and images that are presented to the user by the controller.

The **Controller(s)** are responsible for serving the proper views (typically some type of HTML with template code embedded in them or JSPs). It's also responsible for having access to the proper service APIs to retrieve data when a user asks for it.

The **Model(s)** in the pattern represents all of the object models related to the business logic. These models may be simple POJOs or complex, abstract class structures. The complexity all depends on what business process is being modeled.

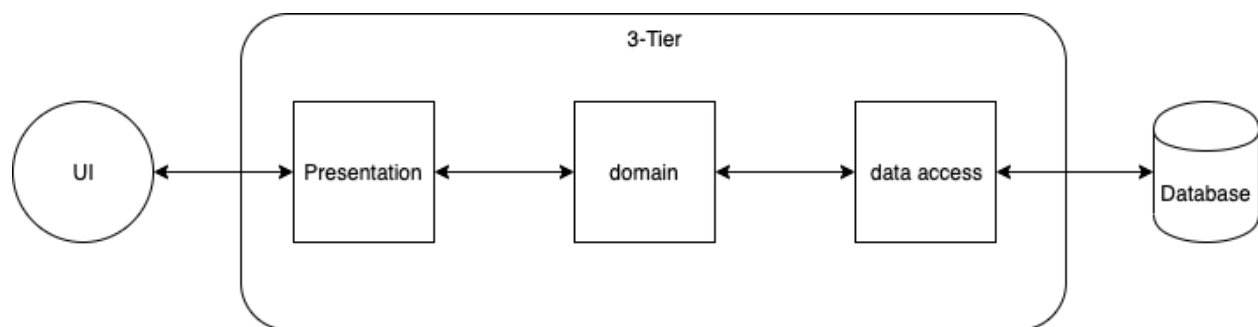


## 3-Tier

In addition to MVC, there is another pattern that is commonly used in web applications; it's something called the **3-Tier pattern**. This is just a more specific type of an N-Tier application, but its structure is pretty simple to understand.

The pattern consists of 3 layers or tiers. The first is the presentation layer, which is usually some type of controller or servlet. It's job it is to route requests to the proper service and handle UI calls. The next layer is the business, application, or domain layer (depending on the engineer you talk to). This is usually implemented with some kind of service class and is responsible for all of the business logic we may have in our application. It's also responsible for calling the appropriate methods from the data access layer. The last layer is the data access layer. This layer is responsible for communications with the database. When a query is run, it's this layer that's responsible for marshalling it and sending the query to the database.

The pattern looks something like this:



<https://www.baeldung.com/spring-mvc-tutorial>

MVC

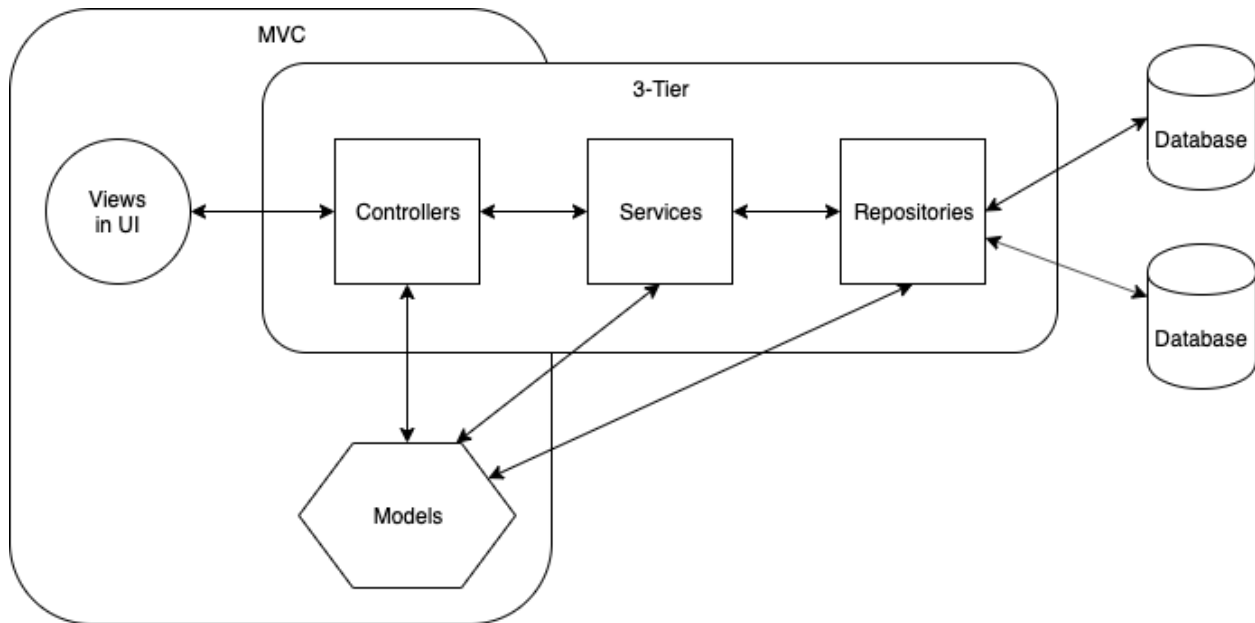


<https://www.izenda.com/5-benefits-3-tier-architecture/>

3-Tier

## Merging Patterns

In reality, these 2 patterns are often used together and do, in-fact, work well together. When implementing these 2 patterns, we end up with a fully functional web application that has great separation of concerns (yet another pattern) as well as high maintainability and code reusability.



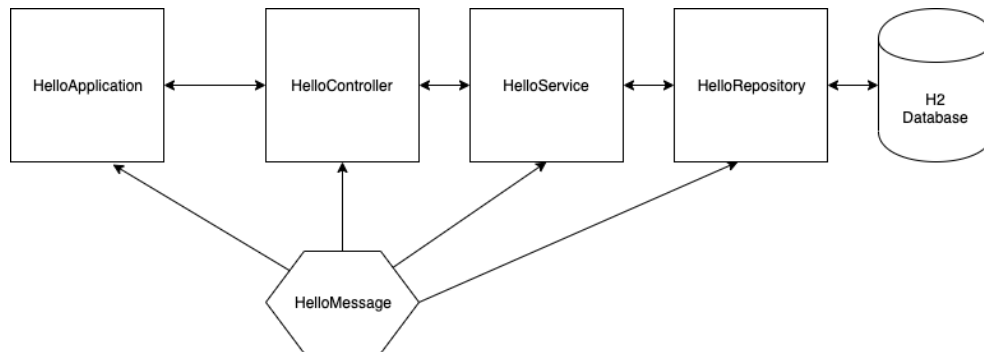
This set of patterns is useful because it scales well, which means it works great for small as well as large enterprise applications. This set of patterns is also used a lot with other architectural patterns such as microservices.



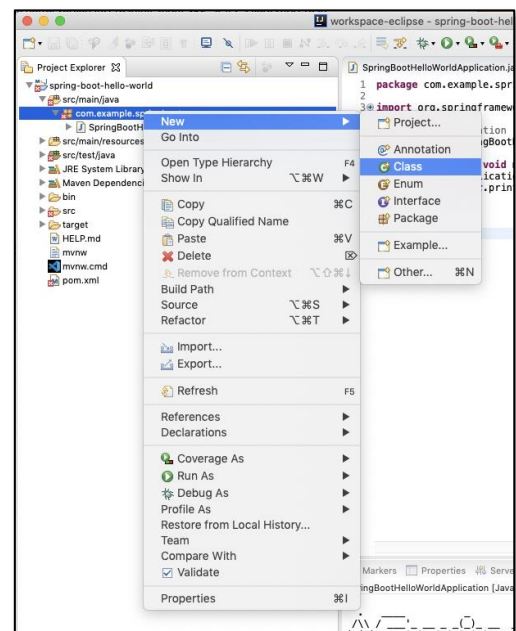
Want to learn more about microservices, check out the recently updated Spring tutorial on it at <https://spring.io/blog/2015/07/14/microservices-with-spring>

## Applying the Patterns

So, let's try these patterns out by adding a few classes to our application to make it more useful. We're going to create a basic **HelloWorld web application** by implementing the two patterns we learned about above and adding a few classes to our existing application.



45. Going back to our Hello World application, let's add a new controller class by right-clicking on our package, then selecting **New** → **Class**.
46. In the wizard that pops up, name the class **HelloController** and click **Finish**.
47. Repeat the same steps to create new classes for **HelloService** and **HelloRepository**.
48. We also need a model for our message, so let's create a new class called **HelloMessage**.
49. Let's also change the name of the driver to something more manageable - right-click on the driver in your package structure then click on **Refactor** → **Rename**. Rename the class to **HelloApplication**. Click **Finish**.



50. Once you're done, you should have a total of 5 classes in your package.

- HelloApplication
- HelloController
- HelloMessage
- HelloRepository
- HelloService

51. Let's start with our model class first since nearly all of the layers interact with our model class at some point.
52. First, let's make the modifications shown below to the `HelloMessage` class. (Hint: Copy/Paste is much faster than typing.)

### HelloMessage

```
package com.example.springboot;

import javax.persistence.*;

@Entity
public class HelloMessage {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String message;

    public HelloMessage() {

    }

    public HelloMessage(String message) {

        this.message = message;
    }

    public Long getId() {

        return id;
    }

    public void setId(Long id) {

        this.id = id;
    }

    public String getMessage() {

        return message;
    }

    public void setMessage(String message) {

        this.message = message;
    }

    @Override
    public String toString() {

        StringBuilder builder = new StringBuilder();
        builder.append(message);
        return builder.toString();
    }

}
```

- The `@Entity` annotation marks the class as a Hibernate entity so it can be saved in the database.
- The `@Id` annotation is required to mark the id field as the primary key in the `HELLO_MESSAGE` table.
- The `@GeneratedValue` annotation tells the database how to increment the id each time a `HelloMessage` object is saved. By default, it starts at 1 and increments by 1.
- The class has two constructors – a default constructor (which is required by the Spring framework) and a constructor that takes a message as a formal parameter (which is used by the service).
- The `toString` method displays the message contents.

53. Next, let's update the repository class. This one is simple.

#### HelloRepository

```
package com.example.springboot;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface HelloRepository extends CrudRepository<HelloMessage, Long> {
}
```

- The `@Repository` annotation is one of the many convenience annotations in Spring and does a couple things for us. First, and most obvious, it marks this class as a repository class. This also has the effect of making this a managed bean in Spring, which allows us to inject it into classes where we'll need to call methods on it.
- You'll notice this class is now an interface. This interface extends `CrudRepository`, which is a generic class that has a few common methods declared already for us including a way to save a record and a way to retrieve all records from the table. We can also pull records by id. The interface defines what types of objects will be held in the table related to it (`HelloMessage`) as well as the type of the primary key (`Long`).
- The JPA framework uses a pattern called **Coding to Interfaces**, which is just a specific implementation of basic inheritance patterns. This allows us to gain the use of pre-implemented code without knowing what the implementation classes are.



There are ways in JPA to create custom queries using special method naming conventions. For a deep dive into these concepts, check out the official JPA documentation at <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories>

54. Next, we'll update the service class.

### HelloService

```
package com.example.springboot;

import org.springframework.stereotype.Service;

@Service
public class HelloService {

    private HelloRepository repository;

    public HelloService(HelloRepository repository) {

        this.repository = repository;
    }

    public String getMessage() {

        // Database returns an Iterable
        Iterable<HelloMessage> messages = this.repository.findAll();
        String message = "";

        // If the message is null, we'll return an empty string
        if(messages != null && messages.iterator().hasNext()) {
            message = messages.iterator().next().getMessage();
        }

        return message;
    }

    public String setMessage(String message) {

        // Message to save
        HelloMessage helloMessage = new HelloMessage(message);

        // Same message is echoed back from database on successful save
        HelloMessage returnedMessage = this.repository.save(helloMessage);

        return returnedMessage.getMessage();
    }

}
```

- The `@Service` annotation is another convenience annotation that marks this class so it gets picked up and managed by Spring.
- This class uses Spring dependency injection to inject a reference to the repository through the constructor. The fact that there is a managed bean with the same name as the field type tells Spring we want that class injected into this one.
- There are two methods in this class which are used to get and set a message into the database. More methods can be added as needed, such as a method to get a message by id.
- The `getMessage` method calls the `findAll` method in the repository, which returns an **Iterable** object. From that, once we do some sanity checking (Is the object null? Does

the object have any records in it?), we can then extract the first message from the list of messages returned.

- The **setMessage** method calls the **save** method in the repository. The save method just echoes back the object that was just saved, if successful. This functionality can be used in the controller to create success messages for the user, if needed.

55. Next, let's update the controller.

### HelloController

```
package com.example.springboot;

import org.springframework.stereotype.Controller;

@Controller
public class HelloController {

    private HelloService service;

    public HelloController(HelloService service) {

        this.service = service;
    }

    public String getMessage() {

        return this.service.getMessage();
    }

    public String setMessage(String message) {

        return this.service.setMessage(message);
    }

}
```

- This controller is pretty simple.
- The `@Controller` annotation is yet another convenience annotation that marks this so it gets picked up by Spring when the Spring container initializes. This allows us to get a reference to it by name from the driver class.
- This class also uses the constructor injection pattern for dependency injection, just as the service did. As an option, you'll also see this pattern used with the constructor being marked with a `@Autowired` annotation. This annotation used to be required for this pattern, but is no longer needed and is omitted to keep the code cleaner and clearer.
- The controller has two methods, a `getMessage` and `setMessage` method. Each of these uses the reference the controller has to the service to call its methods, which, in-turn, call the repository. Classic 3-tier pattern.

56. Next, let's update the driver class.

### HelloApplication

```
package com.example.springboot;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class HelloApplication {

    // create a logger
    private static final Logger logger = LoggerFactory.getLogger(HelloApplication.class.getName());

    public static void main(String[] args) {

        // Get a reference to the Spring context
        ConfigurableApplicationContext context = SpringApplication.run(HelloApplication.class, args);

        // Get a reference to the HelloController, which is a managed Bean
        HelloController controller = (HelloController) context.getBean("helloController");

        // save a message
        controller.setMessage("Hello Spring!");

        // retrieve the message
        String message = controller.getMessage();

        // Output message
        logger.info(message);

    }
}
```

- We already talked about the `@SpringBootApplication` annotation, but by way of review, remember it encapsulates 3 other annotations - `@Configuration` (through a convenience annotation - `@SpringBootConfiguration`), `@EnableAutoConfiguration` and `@ComponentScan`.
- A logger is created so we can output our message on a standard log line. We're using the SLF4J library, which is included transitively in several of the dependencies, including the JPA dependency.
- The run method has been modified so we can capture a reference to the Spring context when it is called. This allows us to then get a reference to the `HelloController` managed bean that has had the proper dependency injection done to it. If we were to instantiate a `HelloController` like we normally do, we'd get a `NullPointerException` when trying to run the code because the service reference in the controller would be null. (That's assuming we add a default constructor to our controller.)



- The driver saves a message first with the *setMessage* method, then it gets the same message using the *getMessage* method.
- The driver then uses our logger to output the message to the console. (In all cases, we never use `System.out` or `System.err` to print log messages. We always use loggers in enterprise level software.)

57. Lastly, we need to add some configuration to our `application.properties` file. This file is located under `src/main/resources`.

### `application.properties`

```
# Server settings
server.port=8080

#####
# H2 Settings
#####
# H2 console can be accessed at http://localhost:8080/h2-console
#####
# Login
#####
# Driver Class: org.h2.Driver
# JDBC URL: jdbc:h2:mem:testdb
# User Name: sa
# Password:
#####
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# Hibernate & JPA settings
spring.jackson.serialization.indent_output=true
spring.jackson.serialization.fail-on-empty-beans=false
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

- `application.properties` files are just properties files that have key/value pairs in them. They are automatically picked up by Spring if they are in the resources folder.
- `server.port` is used to set the port your application runs on. If you have a conflict with 8080, you can start on a different port by changing this value. The default for this is 8080 if not specified.
- `spring.h2.console.enabled` enables the H2 console so you can look directly at your data in the H2 database. Other databases have their own utilities for looking at data. There are also IDE native tools for this as well as 3rd party tools.
- `spring.h2.console.path` sets the path for the root of the h2 console.
- `spring.jackson.serialization.indent_output` affects the way JSON is displayed in the browser.

- `spring.jackson.serialization.fail-on-empty-beans` turns off the automatic failure of your app when no data is returned from the database. Typically, we want this to be false, although in more robust applications, you can handle the exception that is normally thrown.
- `spring.jpa.hibernate.ddl-auto` sets the behavior of JPA when starting up. Valid values are:
  - none
  - validate
  - update
  - create
  - create-drop

See the official documentation for more details at

<https://docs.spring.io/spring-boot/docs/current/reference/html/howto-database-initialization.html>

- `spring.jpa.show-sql` is a Boolean toggle that turns on and off the SQL queries in the log. Normally we turn this off, but during app development, we may need to see what Hibernate is up to when calling our database. That's when we turn this on.
- `spring.jpa.properties.hibernate.format_sql` is another Boolean toggle that formats the log SQL to be human readable. Normally when we want to look at the SQL in the logs, we need this set to true to be able to read them.

58. If your application is still running, shut it off from the console tab by selecting the red button.

59. We need to do a Maven build again, so right-click on the parent project node in the Project Explorer then choose **Run-As → Maven Install**.

60. Once you get the Build Success message, run your application by right-clicking anywhere on your driver code then select **Run-As → Java Application**. You should get console output similar to below.

```

HelloApplication [Java Application] /Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java (Oct 2, 2019, 5:16:25 PM)

:: Spring Boot :: (v2.1.8.RELEASE)

2019-10-02 17:16:26.128 INFO 35058 --- [ restartedMain] com.example.springboot.HelloApplication : Starting HelloApplication on MACLTUS66706 with PID 35058 (/Volum
2019-10-02 17:16:26.130 INFO 35058 --- [ restartedMain] com.example.springboot.HelloApplication : No active profile set, falling back to default profiles: default
2019-10-02 17:16:26.167 INFO 35058 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : DevTools property defaults active! Set 'spring.devtools.add-prop
2019-10-02 17:16:26.167 INFO 35058 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging
2019-10-02 17:16:26.807 INFO 35058 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data repositories in DEFAULT mode.
2019-10-02 17:16:26.857 INFO 35058 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 42ms. Found 1 reposi
2019-10-02 17:16:27.208 INFO 35058 --- [ restartedMain] trationDelegatesBeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactio
2019-10-02 17:16:27.474 INFO 35058 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-10-02 17:16:27.496 INFO 35058 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-10-02 17:16:27.496 INFO 35058 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.24]
2019-10-02 17:16:27.584 INFO 35058 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-10-02 17:16:27.584 INFO 35058 --- [ restartedMain] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1417 ms
2019-10-02 17:16:28.005 INFO 35058 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2019-10-02 17:16:28.145 INFO 35058 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2019-10-02 17:16:28.197 INFO 35058 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
name: default
...
2019-10-02 17:16:28.251 INFO 35058 --- [ restartedMain] org.hibernate.Version : HHH000412: Hibernate Core {5.3.11.Final}
2019-10-02 17:16:28.252 INFO 35058 --- [ restartedMain] org.hibernate.cfg.Environment : HHH000286: hibernate.properties not found
2019-10-02 17:16:28.527 INFO 35058 --- [ restartedMain] org.hibernate.annotations.common.Version : HHCAN000001: Hibernate Commons Annotations {5.0.4.Final}
2019-10-02 17:16:28.523 INFO 35058 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect

Hibernate:
drop table hello_message if exists
Hibernate:
drop sequence if exists hibernate_sequence
Hibernate: create sequence hibernate_sequence start with 1 increment by 1
Hibernate:
create table hello_message (
id bigint not null,
message varchar(255),
primary key (id)
)
2019-10-02 17:16:29.061 INFO 35058 --- [ restartedMain] o.h.t.schema.internal.SchemaCreatorImpl : HHH000476: Executing import script 'org.hibernate.tool.schema.in
2019-10-02 17:16:29.063 INFO 35058 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2019-10-02 17:16:29.091 INFO 35058 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2019-10-02 17:16:29.527 INFO 35058 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-10-02 17:16:29.577 WARN 35058 --- [ restartedMain] @WebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, databa
2019-10-02 17:16:29.961 INFO 35058 --- [ restartedMain] o.s.b.a.e.web.EndpointLinksResolver : Exposing 2 endpoint(s) beneath base path '/actuator'
2019-10-02 17:16:30.036 INFO 35058 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2019-10-02 17:16:30.038 INFO 35058 --- [ restartedMain] com.example.springboot.HelloApplication : Started HelloApplication in 4.128 seconds (JVM running for 4.487

Hibernate:
call next value for hibernate_sequence
Hibernate:
insert
into
hello_message
(message, id)
values
(?, ?)
2019-10-02 17:16:30.131 INFO 35058 --- [ restartedMain] o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
Hibernate:
select
hellomessa0_.id as id1_0_,
hellomessa0_.message as message2_0_
from
hello_message hellomessa0_
2019-10-02 17:16:30.208 INFO 35058 --- [ restartedMain] com.example.springboot.HelloApplication : Hello Spring!

```

- As you can see, Hibernate created a table called hello\_message for us and inserted a record using what's called a prepared statement. These are similar to regular SQL, except there are placeholders for the values inserted. This type of query is much more secure than a regular SQL statement.
- At the end of the log, we can see the message that was retrieved from the database.

Let's take a look at our database now.

61. Navigate to <http://localhost:8080/h2-console>

62. Ensure the **JDBC URL** is the same as below, then click **Test Connection**. You should get a Test successful message. Click on **Connect**.

**Login**

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

---

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:testdb

User Name: sa

Password:

Connect Test Connection

**Test successful**

63. In the next screen, click on the **HELLO\_MESSAGE** table in the left column. If you did it correctly, a pre-made query will pop up in the box to the right.

64. Click on **Run**.

65. You should see 1 record in the table below the query.

Auto commit Max rows: 1000 Auto complete: Off Auto select: On

jdbc:h2:mem:testdb

HELLO\_MESSAGE

INFORMATION\_SCHEMA

Sequences

Users

H2 1.4.199 (2019-03-13)

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM HELLO\_MESSAGE

SELECT \* FROM HELLO\_MESSAGE;

ID	MESSAGE
1	Hello Spring!

(1 row, 3 ms)

Edit

This demonstrates that our application is working properly and persisting data in the H2 database.

## Let's Review

---

- In this section, we set up a full web application using the MVC and 3-Tier patterns.
- We learned about several Spring convenience annotations, including @Controller, @Service and @Repository.
- We learned about a few JPA annotations - @Entity, @Id and @GeneratedValue that are used on model objects.
- We learned we didn't need to implement any code to use JPA queries – we had several already made for us right out of the box.
- We learned how to get a reference to a managed bean from the Spring context as well as how to get a reference to the Spring context itself.
- We learned how to use an application.properties file.
- We learned about several different JPA properties that affect how the database initializes and how it displays data in the log.

Whew! That's a lot!

Awesome!

## Knowledge Check

---

- Which 3 convenience annotations did we learn about?
- Which of these annotations marks a class so it can be managed as a Spring bean?
- Which annotation is used to mark a class as a JPA/Hibernate entity?
- Which annotation marks a field in a class as its primary key in the database table associated with that model?
- How do we find out, within our IDE, what type of object is returned from any given method in our code?
- What is a transitive dependency?



The next phase of the Hello World Spring Boot application will be an easy one. We need only make modifications to one of our classes to turn our application into a REST service, which we'll then be able to query with our browser.

66. Stop your application by clicking on the red button in the Console tab.

67. Modify the **HelloController** as follows:

### HelloController

```
package com.example.springboot;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/")
public class HelloController {

    private HelloService service;

    public HelloController(HelloService service) {

        this.service = service;
    }

    @GetMapping(value = "getMessage", produces = "application/json")
    public String getMessage() {

        return this.service.getMessage();
    }

    @PostMapping(value = "addMessage", produces = "application/json")
    public String setMessage(String message) {

        return this.service.setMessage(message);
    }

}
```

- You'll notice we changed the `@Controller` annotation to `@RestController`. This is another convenience annotation that adds the `@RequestBody` annotation to your controller. This tells spring that any of the return types in this class's methods need to be returned to the user in the HTTP request body.
- We added the `@RequestMapping` annotation. This tells spring that any of the REST endpoints in this controller will be at the root "/" of our web address.

- We added a `@GetMapping` annotation to our `getMessage` method. This annotation is a convenience annotation that encapsulates a `@RequestMapping` annotation set up as a GET HTTP endpoint.
- We added a `@PostMapping` annotation to our `setMessage` method. This annotation is a convenience annotation that encapsulates a `@RequestMapping` annotation set up as a POST HTTP endpoint.
- The `produces` attribute on both the `@GetMapping` and `@PostMapping` annotations is used to tell Spring we want the return value marshalled into a JSON (JavaScript Object Notation) formatted string.



If you don't know what JSON is, there are a lot of very good tutorials on it, including:

[https://www.w3schools.com/js/js\\_json\\_intro.asp](https://www.w3schools.com/js/js_json_intro.asp)

<https://www.baeldung.com/spring-boot-json>

Now let's run our application and take a look at the data in the browser.

68. Save your changes.

69. We need to do a Maven build again, so right-click on the parent project node in the Project Explorer then choose **Run-As** → **Maven Install**.

70. Once you get the Build Success message, run your application by right-clicking anywhere on your driver code then select **Run-As** → **Java Application**. You should get console output similar to below.

```

2019-10-02 18:05:18.962 INFO 35240 --- [ restartedMain] com.example.springboot.HelloApplication : Starting HelloApplication on MACLTUS66786 with PID 35240 (/volu
2019-10-02 18:05:18.964 INFO 35240 --- [ restartedMain] com.example.springboot.HelloApplication : No active profile set, falling back to default profiles: default
2019-10-02 18:05:19.003 INFO 35240 --- [ restartedMain] e.DevToolsPropertyDefaultsPostProcessor : DevTools property defaults active! Set 'spring.devtools.add-prop
2019-10-02 18:05:19.004 INFO 35240 --- [ restartedMain] e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging
2019-10-02 18:05:19.726 INFO 35240 --- [ restartedMain] s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data repositories in DEFAULT mode.
2019-10-02 18:05:19.791 INFO 35240 --- [ restartedMain] s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 57ms. Found 1 reposi
2019-10-02 18:05:20.237 INFO 35240 --- [ restartedMain] t.transaction.annotation.ProxyTransaction : Bean 'org.springframework.transaction.annotation.ProxyTransaction
2019-10-02 18:05:20.524 INFO 35240 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-10-02 18:05:20.548 INFO 35240 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-10-02 18:05:20.548 INFO 35240 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.24]
2019-10-02 18:05:20.688 INFO 35240 --- [ restartedMain] o.s.c.c.c.(Tomcat).localhost.[] : Initializing Spring embedded WebApplicationContext
2019-10-02 18:05:20.688 INFO 35240 --- [ restartedMain] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1684 ms
2019-10-02 18:05:21.098 INFO 35240 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2019-10-02 18:05:21.237 INFO 35240 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2019-10-02 18:05:21.288 INFO 35240 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper : HHH000284: Processing PersistenceUnitInfo [
name: default
...]
2019-10-02 18:05:21.332 INFO 35240 --- [ restartedMain] org.hibernate.Version : HHH0000412: Hibernate Core (5.3.11.Final)
2019-10-02 18:05:21.333 INFO 35240 --- [ restartedMain] org.hibernate.cfg.Environment : HHH0000285: hibernate.properties not found
2019-10-02 18:05:21.449 INFO 35240 --- [ restartedMain] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations (5.0.4.Final)
2019-10-02 18:05:21.579 INFO 35240 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
Hibernate:
drop table hello_message if exists
Hibernate:
drop sequence if exists hibernate_sequence
Hibernate: create sequence hibernate_sequence start with 1 increment by 1
Hibernate:
create table hello_message (
id bigint not null,
message varchar(255),
primary key (id)
)
2019-10-02 18:05:22.133 INFO 35240 --- [ restartedMain] o.h.t.schema.internal.SchemaCreatorImpl : HHH0000476: Executing import script 'org.hibernate.tools.schema.in
2019-10-02 18:05:22.135 INFO 35240 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default
2019-10-02 18:05:22.168 INFO 35240 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2019-10-02 18:05:22.555 INFO 35240 --- [ restartedMain] o.s.i.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-10-02 18:05:22.599 WARN 35240 --- [ restartedMain] aWebConfigurationsJpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, databa
2019-10-02 18:05:23.025 INFO 35240 --- [ restartedMain] o.s.b.a.e.web.EndpointLinksResolver : Exposing 2 endpoint(s) beneath base path '/actuator'
2019-10-02 18:05:23.104 INFO 35240 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path '/'
2019-10-02 18:05:23.106 INFO 35240 --- [ restartedMain] com.example.springboot.HelloApplication : Started HelloApplication in 4.363 seconds (JVM running for 4.665
Hibernate:
call next value for hibernate_sequence
Hibernate:
insert
into
hello_message
(message, id)
values
(?, ?)
2019-10-02 18:05:23.197 INFO 35240 --- [ restartedMain] o.h.h.l.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
Hibernate:
select
hellomessa0_.id as id1_0_,
hellomessa0_.message as message2_0_
from
hello_message hellomessa0_
2019-10-02 18:05:23.288 INFO 35240 --- [ restartedMain] com.example.springboot.HelloApplication : Hello Spring!

```

You'll notice there's not a huge difference in the console output from what we saw before. The biggest change to the application is in the way we can access our new REST endpoints now.

71. Go to your browser and navigate to <http://localhost:8080/getMessage>

72. You should see a message that says "Hello Spring!". This may not look like JSON, but it is. That's the way a single String is displayed in JSON format in a browser. Once you start working with JSON more, you'll see more complex structures.



## Let's Review

---

- In this section, we learned about how to set up a REST service using Spring and Spring annotations.
- We learned what the @RestController annotation does and what it brings to the party.
- We learned what the @RequestMapping does and how to set a base URL for the controller.
- We learned what the @GetMapping and @PostMapping annotations do for our controller methods.
- We learned how to set URLs for each of our REST endpoints.
- We learned about how JSON is used to send data to a browser.

## Knowledge Check

---

- What annotation is used to mark a controller as having one or more REST endpoints?
- What annotation is used to set the base URL for a controller?
- What annotations are used to specify what type of data is sent back to the user?
- What annotations are used to specify what URLs are used for each endpoint?
- What does JSON stand for?



- In this series of tutorials and demos, you learned about Spring and Spring Boot.
- You learned why we would want to use these frameworks.
- You learned how to set up a Spring Boot application from scratch.
- You learned how to introduce common patterns in a systematic way.
- You learned how to run a Spring Boot web application in eclipse.
- You learned how to create a RESTful web service using Spring Boot.
- And you learned how to have fun doing it all!

# That's all folks!

---

## Additional Resources

---

This is a list of additional resources available to expand on your Spring Boot application.



For information on how to auto generate a full Angular or React Spring Boot application, check out the JHipster generator at:

<https://jhipster.tech>



For information on Angular, check out the Angular tutorial on their home site at:

<https://angular.io/start>



For information on React, check out the React tutorial at:  
<https://reactjs.org/tutorial/tutorial.html>



For information on a very good messaging system you can use with microservices, check out the RabbitMQ tutorials for Java at:

<https://www.rabbitmq.com/getstarted.html>

Just choose Java or Spring AMQP in the menu. Spring AMQP is one of the Spring projects and works well with Spring Boot.



## Check 1

---

- What Spring framework allows you to monitor your application?
  - Spring Actuator
- What is the biggest benefit to using Spring?
  - Dependency Injection
- What are the 3 annotations that are encapsulated by the @SpringBootApplication annotation?
  - @Configuration
  - @EnableAutoConfiguration
  - @ComponentScan

## Check 2

---

- Which 3 convenience annotations did we learn about?
  - @Controller
  - @Service
  - @Repository
- Which of these annotations marks a class so it can be managed as a Spring bean?
  - @Controller
  - @Service
  - @Repository
- Which annotation is used to mark a class as a JPA/Hibernate entity?
  - @Entity
- Which annotation marks a field in a class as its primary key in the database table associated with that model?
  - @Id
- How do we find out, within our IDE, what type of object is returned from any given method in our code?
  - Ctrl-Click on the method name.
- What is a transitive dependency?
  - It's a dependency that one of your direct dependencies has.

## Check 3

---

- What annotation is used to mark a controller as having one or more REST endpoints?
  - @RestController
- What annotation is used to set the base URL for a controller?
  - @RequestMapping
- What annotations are used to specify what type of data is sent back to the user?
  - @GetMapping
  - @PostMapping
- What annotations are used to specify what URLs are used for each endpoint?
  - @GetMapping
  - @PostMapping
- What does JSON stand for?
  - JavaScript Object Notation