Spring Boot



Spring Boot Tutorial provides basic and advanced concepts of Spring Framework. Our Spring Boot Tutorial is designed for beginners and professionals both.

Spring Boot is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring framework.

Our Spring Boot Tutorial includes all topics of Spring Boot such, as features, project, maven project, starter project wizard, Spring Initializr, CLI, applications, annotations, dependency management, properties, starters, Actuator, JPA, JDBC, etc.

What is Spring Boot

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

25.9M

525

Java Try Catch

It is a Spring module that provides the **RAD (*Rapid Application Development*)** feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.



In short, Spring Boot is the combination of **Spring Framework** and **Embedded Servers**.

In Spring Boot, there is no requirement for XML configuration (deployment descriptor). It uses convention over configuration software design paradigm that means it decreases the effort of the developer.

We can use Spring **STS IDE** or **Spring Initializr** to develop Spring Boot Java applications.

**Why should we use Spring Boot Framework?**

We should use Spring Boot Framework because:

* The dependency injection approach is used in Spring Boot.
* It contains powerful database transaction management capabilities.
* It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.
* It reduces the cost and development time of the application.

Along with the Spring Boot Framework, many other Spring sister projects help to build applications addressing modern business needs. There are the following Spring sister projects are as follows:

* **Spring Data:** It simplifies data access from the relational and **NoSQL** databases.
* **Spring Batch:** It provides powerful **batch** processing.
* **Spring Security:** It is a security framework that provides robust **security** to applications.
* **Spring Social:** It supports integration with **social networking** like LinkedIn.
* **Spring Integration:** It is an implementation of Enterprise Integration Patterns. It facilitates integration with other **enterprise applications** using lightweight messaging and declarative adapters.

Advantages of Spring Boot

* It creates **stand-alone** Spring applications that can be started using Java **-jar**.
* It tests web applications easily with the help of different **Embedded** HTTP servers such as **Tomcat, Jetty,** etc. We don't need to deploy WAR files.
* It provides opinionated '**starter**' POMs to simplify our Maven configuration.
* It provides **production-ready** features such as **metrics, health checks,** and **externalized configuration**.
* There is no requirement for **XML** configuration.
* It offers a **CLI** tool for developing and testing the Spring Boot application.
* It offers the number of **plug-ins**.
* It also minimizes writing multiple **boilerplate codes** (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.
* It **increases productivity** and reduces development time.

Limitations of Spring Boot

Spring Boot can use dependencies that are not going to be used in the application. These dependencies increase the size of the application.

Goals of Spring Boot

The main goal of Spring Boot is to reduce **development, unit test,** and **integration test** time.

* Provides Opinionated Development approach
* Avoids defining more Annotation Configuration
* Avoids writing lots of import statements
* Avoids XML Configuration.

By providing or avoiding the above points, Spring Boot Framework reduces **Development time, Developer Effort,** and **increases productivity**.

Prerequisite of Spring Boot

To create a Spring Boot application, following are the prerequisites. In this tutorial, we will use **Spring Tool Suite** (STS) IDE.

* Java 1.8
* Maven 3.0+
* Spring Framework 5.0.0.BUILD-SNAPSHOT
* An IDE (Spring Tool Suite) is recommended.

Spring Boot Features

* Web Development
* SpringApplication
* Application events and listeners
* Admin features
* Externalized Configuration
* Properties Files
* YAML Support
* Type-safe Configuration
* Logging
* Security

**Web Development**

It is a well-suited Spring module for web application development. We can easily create a self-contained HTTP application that uses embedded servers like **Tomcat, Jetty,** or Undertow. We can use the **spring-boot-starter-web** module to start and run the application quickly.

**SpringApplication**

The SpringApplication is a class that provides a convenient way to bootstrap a Spring application. It can be started from the main method. We can call the application just by calling a static run() method.

1. **public** **static** **void** main(String[] args)
2. {
3. SpringApplication.run(ClassName.**class**, args);
4. }

**Application Events and Listeners**

Spring Boot uses events to handle the variety of tasks. It allows us to create factories file that is used to add listeners. We can refer it to using the **ApplicationListener key**.

Always create factories file in META-INF folder like **META-INF/spring.factories**.

**Admin Support**

Spring Boot provides the facility to enable admin-related features for the application. It is used to access and manage applications remotely. We can enable it in the Spring Boot application by using **spring.application.admin.enabled** property.

**Externalized Configuration**

Spring Boot allows us to externalize our configuration so that we can work with the same application in different environments. The application uses YAML files to externalize configuration.

**Properties Files**

Spring Boot provides a rich set of **Application Properties**. So, we can use that in the properties file of our project. The properties file is used to set properties like **server-port =8082** and many others. It helps to organize application properties.

**YAML Support**

It provides a convenient way of specifying the hierarchical configuration. It is a superset of JSON. The SpringApplication class automatically supports YAML. It is an alternative of properties file.

**Type-safe Configuration**

The strong type-safe configuration is provided to govern and validate the configuration of the application. Application configuration is always a crucial task which should be type-safe. We can also use annotation provided by this library.

**Logging**

Spring Boot uses Common logging for all internal logging. Logging dependencies are managed by default. We should not change logging dependencies if no customization is needed.

**Security**

Spring Boot applications are spring bases web applications. So, it is secure by default with basic authentication on all HTTP endpoints. A rich set of Endpoints is available to develop a secure Spring Boot application.

Spring Boot Version

The latest version of Spring Boot is **2.0**. It introduces a lot of new features along with some modifications and replacement.

Spring Boot 2.0

Let's have a sneak peek at Spring Boot 2.0.

* **What's New**  
            Infrastructure Upgrade  
            Spring Framework 5
* **What's Changed**  
            Configuration Properties  
            Gradle Plugin  
            Actuators endpoints
* **What's Evolving**  
            Security  
            Metrics

The pivotal team has upgraded the **infrastructure** in which the following tools are involved:

* Supports **Java 8** or above versions
* Supports Apache **Tomcat 8** or above versions
* Supports **Thymeleaf 3**
* Supports **Hibernate 5.2**

In **Spring Framework 5**, the Pivotal team upgraded the following:

* Reactive Spring
  1. Servlet stack
     1. Servlet Container
     2. Servlet API
     3. Spring MVC
  2. **Reactive Stack**
     1. Netty, Servlet 3.1, Undertow
     2. Reactive HTTP Layer
     3. Spring WebFlux
* Functional API
* Kotlin Support

The latest version of Spring Boot is 2.2.1. This release of Spring Boot includes 110 fixes, dependency upgrades, and improvements.

In the Spring Boot v2.2.1, the annotation **@ConfigurationProperties** scanning is now disabled by default. We need to be explicitly opted into by adding the **@ConfigurationPropertiesScan** annotation.

New Features

* Support constructor binding for property nested inside a JavaBean
* Add config property for CodecConfigurer.maxInMemorySize in WebFlux
* Make test slices' type exclude filters public
* Support amqps:// URIs in spring.rabbitmq.addresses

Dependency upgrades

Some dependencies have been upgraded in Spring Boot v2.2.1 are as follows:

* Mongodb 3.11.2
* Spring Security 5.2.1.RELEASE
* Slf4j 1.7.29
* Spring Hateoas 1.0.1.RELEASE
* Hibernate Validator 6.0.18.Final
* Hibernate 5.4.8.Final
* Jetty 9.4.22.v20191022
* Spring Framework 5.2.1
* Spring AMQP 2.2.1
* H2 1.4.200
* Spring Security 5.2
* Spring Batch 4.2

Some important and widely used third-party dependencies are upgraded in this release are as follows:

* Micrometer 1.3.1
* Flyway 6.0.7
* Elasticsearch 6.8.4
* JUnit 5.5
* Jackson 2.10

Performance Improvements

In Spring Boot 2.2.1 the following performance has been improved:

**Lazy Initialization**

In Spring Boot 2.2.1, we can enable global lazy initialization by using the property **spring.main.lazy-initialization** property. It reduces the application startup time.

**Java 13 Support**

Spring Boot 2.2.1 now supports the latest version of Java that is Java 13.

**Immutable Binding**

In the newer version of Spring Boot, Configuration properties support constructor-based binding. The class annotates with**@ConfigurationProperties**annotation is to be immutable. It can be enabled by adding an annotation **@ConfugurationProperties** to a class or one of its constructors with **@ConstructorBinding.**

**RSocket Support**

It is a part of **Spring Security**. RSocket integration is auto-configured when an application finds **spring-security-rsocket** is present on the classpath.

Deprecations in Spring Boot 2.2

* The property **logging.file** has renamed to logging.file.name.
* The property **logging.path** has renamed to logging.file.path.
* The server.connection-timeout property has been deprecated in favor of server-specific properties.
* Joda time support is deprecated in favor of java.time.

**The following improvements are made in the Spring Boot 2.2.1**

* **Java:** Spring Boot 2.2.1
* **Spring Framework 5.2:** This release of Spring Boot upgrades to Spring Framework to 5.2.
* **JMX is disabled:** In this version, JMX is not enabled by default. We can enable it by using the property **jmx.enabled=true**. If you are using the IDE feature to monitor your application, we need to enable it.
* **Fork enabled by default:** Spring Boot application that ran by Maven Plugin is now forked by default.
* **JUnit 5:** Spring Boot v2.2.1 provides **JUnit 5**by default. JUnit 5's vintage engine is also included by default that supports existing JUnit 4-based test classes. We can also use JUnit 4 and JUnit 5 based test classes in the same module.
* **AssertJ 3.12:** This release of Spring Boot upgrades to AssertJ 3.12. It contains a breaking API changes for assertions related to Iterator.
* **Hibernate Dialect:** In the newer version of Spring Boot, Hibernate chose the dialect to use rather than applying a default dialect based on the detected database.
* **Gradle Requirements:** The latest version of Spring Boot requires Gradle 4.10.

# Spring vs. Spring Boot vs. Spring MVC

## Spring vs. Spring Boot

**Spring:** Spring Framework is the most popular application development framework of Java. The main feature of the Spring Framework is **dependency Injection** or **Inversion of Control** (IoC). With the help of Spring Framework, we can develop a **loosely** coupled application. It is better to use if application type or characteristics are purely defined.

**Spring Boot:** Spring Boot is a module of Spring Framework. It allows us to build a stand-alone application with minimal or zero configurations. It is better to use if we want to develop a simple Spring-based application or RESTful services.

The primary comparison between Spring and Spring Boot are discussed below:

|  |  |
| --- | --- |
| **Spring** | **Spring Boot** |
| **Spring Framework** is a widely used Java EE framework for building applications. | **Spring Boot Framework** is widely used to develop **REST APIs**. |
| It aims to simplify Java EE development that makes developers more productive. | It aims to shorten the code length and provide the easiest way to develop **Web Applications**. |
| The primary feature of the Spring Framework is **dependency injection**. | The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement. |
| It helps to make things simpler by allowing us to develop **loosely coupled** applications. | It helps to create a **stand-alone** application with less configuration. |
| The developer writes a lot of code (**boilerplate code**) to do the minimal task. | It **reduces** boilerplate code. |
| To test the Spring project, we need to set up the sever explicitly. | Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc. |
| It does not provide support for an in-memory database. | It offers several plugins for working with an embedded and **in-memory** database such as **H2**. |
| Developers manually define dependencies for the Spring project in **pom.xml**. | Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement. |

## Spring Boot vs. Spring MVC

**Spring Boot:** Spring Boot makes it easy to quickly bootstrap and start developing a Spring-based application. It avoids a lot of boilerplate code. It hides a lot of complexity behind the scene so that the developer can quickly get started and develop Spring-based applications easily.

**Spring MVC:** Spring MVC is a Web MVC Framework for building web applications. It contains a lot of configuration files for various capabilities. It is an HTTP oriented web application development framework.

Spring Boot and Spring MVC exist for different purposes. The primary comparison between Spring Boot and Spring MVC are discussed below:

|  |  |
| --- | --- |
| **Spring Boot** | **Spring MVC** |
| **Spring Boot** is a module of Spring for packaging the Spring-based application with sensible defaults. | **Spring MVC** is a model view controller-based web framework under the Spring framework. |
| It provides default configurations to build **Spring-powered** framework. | It provides **ready to use** features for building a web application. |
| There is no need to build configuration manually. | It requires build configuration manually. |
| There is **no requirement** for a deployment descriptor. | A Deployment descriptor is **required**. |
| It avoids boilerplate code and wraps dependencies together in a single unit. | It specifies each dependency separately. |
| It **reduces** development time and increases productivity. | It takes **more** time to achieve the same. |

Spring Boot Architecture

Spring Boot is a module of the Spring Framework. It is used to create stand-alone, production-grade Spring Based Applications with minimum efforts. It is developed on top of the core Spring Framework.

Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.

Before understanding the **Spring Boot Architecture**, we must know the different layers and classes present in it. There are **four** layers in Spring Boot are as follows:

* **Presentation Layer**
* **Business Layer**
* **Persistence Layer**
* **Database Layer**



**Presentation Layer:** The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

**Business Layer:** The business layer handles all the **business logic**. It consists of service classes and uses services provided by data access layers. It also performs **authorization** and **validation**.

**Persistence Layer:** The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

**Database Layer:** In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.

Spring Boot Flow Architecture



* Now we have validator classes, view classes, and utility classes.
* Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc. The architecture of Spring Boot is the same as the architecture of Spring MVC, except one thing: there is no need for **DAO** and **DAOImpl** classes in Spring boot.
* Creates a data access layer and performs CRUD operation.
* The client makes the HTTP requests (PUT or GET).
* The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.
* In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.
* A JSP page is returned to the user if no error occurred.

Spring Initializr

**Spring Initializr** is a **web-based tool** provided by the Pivotal Web Service. With the help of **Spring Initializr**, we can easily generate the structure of the **Spring Boot Project**. It offers extensible API for creating JVM-based projects.

It also provides various options for the project that are expressed in a metadata model. The metadata model allows us to configure the list of dependencies supported by JVM and platform versions, etc. It serves its metadata in a well-known that provides necessary assistance to third-party clients.

Spring Initializr Modules

Spring Initializr has the following module:

* **initializr-actuator:** It provides additional information and statistics on project generation. It is an optional module.
* **initializr-bom:** In this module, **BOM** stands for **Bill Of Materials**. In Spring Boot, BOM is a special kind of **POM** that is used to control the **versions** of a project's **dependencies**. It provides a central place to define and update those versions. It provides flexibility to add a dependency in our module without worrying about the versions.  
  Outside the software world, the **BOM** is a list of parts, items, assemblies, and other materials required to create products. It explains **what, how,** and **where** to collect required materials.
* **initializr-docs:** It provides documentation.
* **initializr-generator:** It is a core project generation library.
* **initializr-generator-spring:**
* **initializr-generator-test:** It provides a test infrastructure for project generation.
* **initializr-metadata:** It provides metadata infrastructure for various aspects of the projects.
* **initializr-service-example:** It provides custom instances.
* **initializr-version-resolver:** It is an optional module to extract version numbers from an arbitrary POM.
* **initializr-web:** It provides web endpoints for third party clients.

Supported Interface

* It supports **IDE STS, IntelliJ IDEA Ultimate, NetBeans, Eclipse**. You can download the plugin from [https://github.com/AlexFalappa/nb-springboot](https://github.com/AlexFalappa/nb-springboot" \t "_blank)

. If you are using VSCode, download the plugin from [https://github.com/microsoft/vscode-spring-initializr](https://github.com/microsoft/vscode-spring-initializr" \t "_blank)

.

* Use Custom Web UI [http://start.spring.io](https://start.spring.io/" \t "_blank)

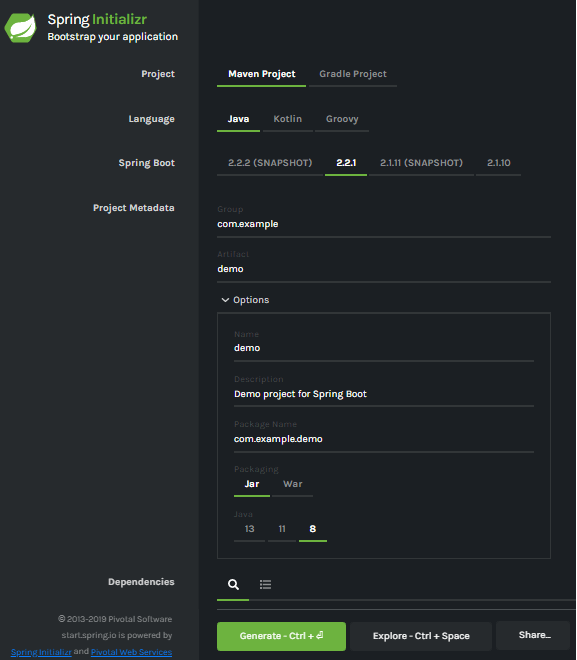
or [https://start-scs.cfapps.io](https://start-scs.cfapps.io/" \t "_blank)

.

* It also supports the command-line with the **Spring Boot CLI** or **cURL** or **HTTPie**.

The following image shows the Spring Initializr UI:

Java Try Catch



Generating a Project

Before creating a project, we must be friendly with UI. Spring Initializr UI has the following labels:

* **Project:** It defines the **kind** of project. We can create either **Maven Project** or **Gradle Project**. We will create a **Maven Project** throughout the tutorial.
* **Language:** Spring Initializr provides the choice among three languages **Java, Kotlin,** and **Groovy**. Java is by default selected.
* **Spring Boot:** We can select the Spring Boot **version**. The latest version is **2.2.2**.
* **Project Metadata:** It contains information related to the project, such as **Group**, Artifact, etc. Group denotes the **package** name; **Artifact** denotes the **Application** name. The default Group name is **com.example**, and the default Artifact name is **demo**.
* **Dependencies:** Dependencies are the collection of artifacts that we can add to our project.

There is another **Options** section that contains the following fields:

* **Name:** It is the same as **Artifact**.
* **Description:** In the description field, we can write a **description** of the project.
* **Package Name:** It is also similar to the **Group** name.
* **Packaging:** We can select the **packing** of the project. We can choose either **Jar** or **War**.
* **Java:** We can select the **JVM** version which we want to use. We will use **Java 8** version throughout the tutorial.

There is a **Generate** button. When we click on the button, it starts packing the project and downloads the **Jar** or **War** file, which you have selected.

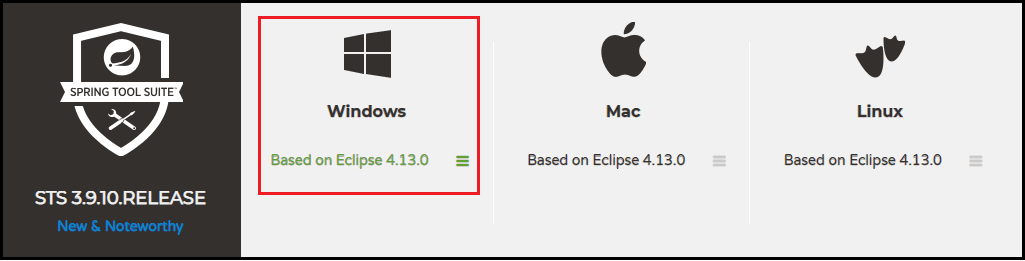
# Download and Install STS IDE

## Spring Tool Suite (STS) IDE

Spring Tool Suite is an IDE to develop Spring applications. It is an Eclipse-based development environment. It provides a ready-to-use environment to implement, run, deploy, and debug the application. It validates our application and provides quick fixes for the applications.

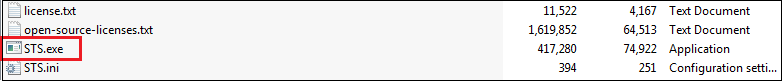
### **Installing STS**

**Step 1:** Download Spring Tool Suite from <https://spring.io/tools3/sts/all>. Click on the platform which you are using. In this tutorial, we are using the Windows platform.

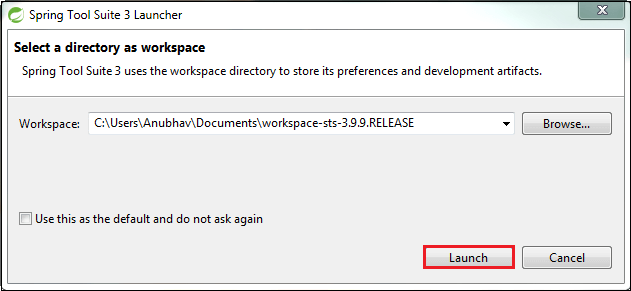


**Step 2:** Extract the **zip** file and install the STS.

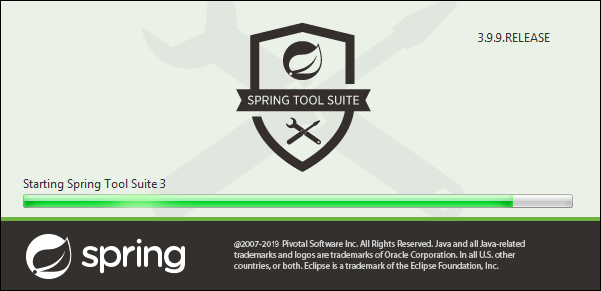
sts-bundle -> sts-3.9.9.RELEASE -> Double-click on the **STS.exe**.



**Step 3:** Spring Tool Suite 3 Launcher dialog box appears on the screen. Click on the **Launch** button. You can change the Workspace if you want.



**Step 4:** It starts launching the STS.



The STS user interface looks like the following:

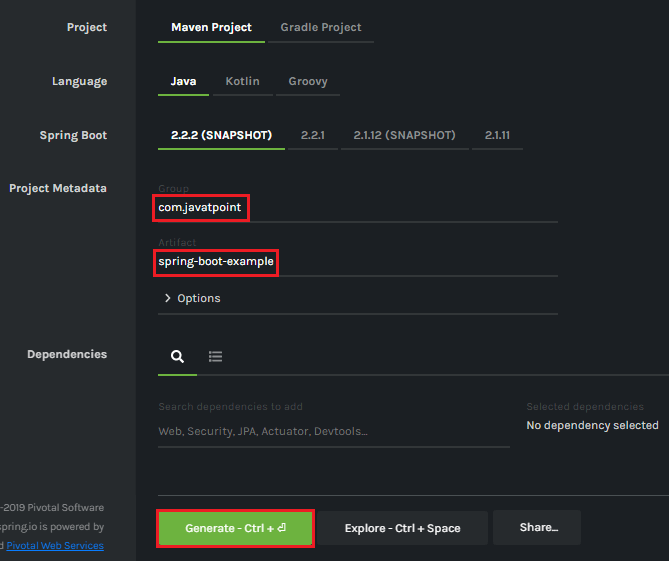
# Download and Install STS IDECreating a Spring Boot Project

Following are the steps to create a simple Spring Boot Project.

**Step 1:** Open the Spring initializr [https://start.spring.io](https://start.spring.io/).

**Step 2:** Provide the **Group** and **Artifact** name. We have provided Group name **com.javatpoint** and Artifact **spring-boot-example**.

**Step 3:** Now click on the **Generate** button.



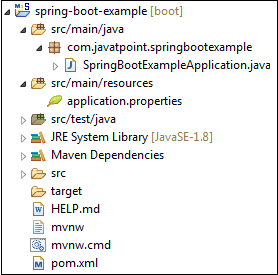
When we click on the Generate button, it starts packing the project in a **.rar** file and downloads the project.

**Step 4:** Extract the **RAR** file.

**Step 5:** **Import** the folder.

File -> Import -> Existing Maven Project -> Next -> Browse -> Select the project -> Finish

It takes some time to import the project. When the project imports successfully, we can see the project directory in the **Package Explorer**. The following image shows the project directory:



**SpringBootExampleApplication.java**

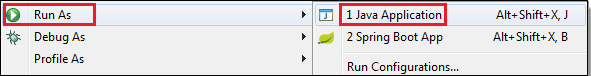
1. **package** com.javatpoint.springbootexample;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootExampleApplication
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(SpringBootExampleApplication.**class**, args);
10. }
11. }

**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 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**org.springframework.boot**</groupId>**
6. **<artifactId>**spring-boot-starter-parent**</artifactId>**
7. **<version>**2.2.2.BUILD-SNAPSHOT**</version>**
8. **<relativePath/>** <!-- lookup parent from repository -->
9. **</parent>**
10. **<groupId>**com.javatpoint**</groupId>**
11. **<artifactId>**spring-boot-example**</artifactId>**
12. **<version>**0.0.1-SNAPSHOT**</version>**
13. **<name>**spring-boot-example**</name>**
14. **<description>**Demo project for Spring Boot**</description>**
15. **<properties>**
16. **<java.version>**1.8**</java.version>**
17. **</properties>**
18. **<dependencies>**
19. **<dependency>**
20. **<groupId>**org.springframework.boot**</groupId>**
21. **<artifactId>**spring-boot-starter**</artifactId>**
22. **</dependency>**
23. **<dependency>**
24. **<groupId>**org.springframework.boot**</groupId>**
25. **<artifactId>**spring-boot-starter-test**</artifactId>**
26. **<scope>**test**</scope>**
27. **<exclusions>**
28. **<exclusion>**
29. **<groupId>**org.junit.vintage**</groupId>**
30. **<artifactId>**junit-vintage-engine**</artifactId>**
31. **</exclusion>**
32. **</exclusions>**
33. **</dependency>**
34. **</dependencies>**
35. **<build>**
36. **<plugins>**
37. **<plugin>**
38. **<groupId>**org.springframework.boot**</groupId>**
39. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
40. **</plugin>**
41. **</plugins>**
42. **</build>**
43. **<repositories>**
44. **<repository>**
45. **<id>**spring-milestones**</id>**
46. **<name>**Spring Milestones**</name>**
47. **<url>**https://repo.spring.io/milestone**</url>**
48. **</repository>**
49. **<repository>**
50. **<id>**spring-snapshots**</id>**
51. **<name>**Spring Snapshots**</name>**
52. **<url>**https://repo.spring.io/snapshot**</url>**
53. **<snapshots>**
54. **<enabled>**true**</enabled>**
55. **</snapshots>**
56. **</repository>**
57. **</repositories>**
58. **<pluginRepositories>**
59. **<pluginRepository>**
60. **<id>**spring-milestones**</id>**
61. **<name>**Spring Milestones**</name>**
62. **<url>**https://repo.spring.io/milestone**</url>**
63. **</pluginRepository>**
64. **<pluginRepository>**
65. **<id>**spring-snapshots**</id>**
66. **<name>**Spring Snapshots**</name>**
67. **<url>**https://repo.spring.io/snapshot**</url>**
68. **<snapshots>**
69. **<enabled>**true**</enabled>**
70. **</snapshots>**
71. **</pluginRepository>**
72. **</pluginRepositories>**
73. **</project>**

**Step 6:** Run the **SpringBootExampleApplication.java** file.

Right-click on the file -> Run As -> Java Applications



The following image shows the application runs successfully.

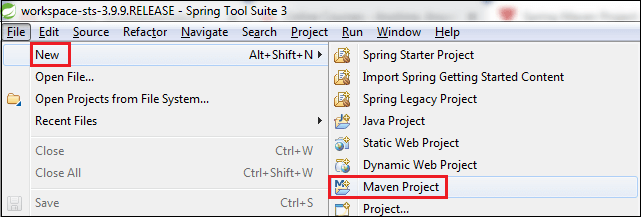
Creating Spring Boot Project

# Creating a Spring Boot Project Using STS

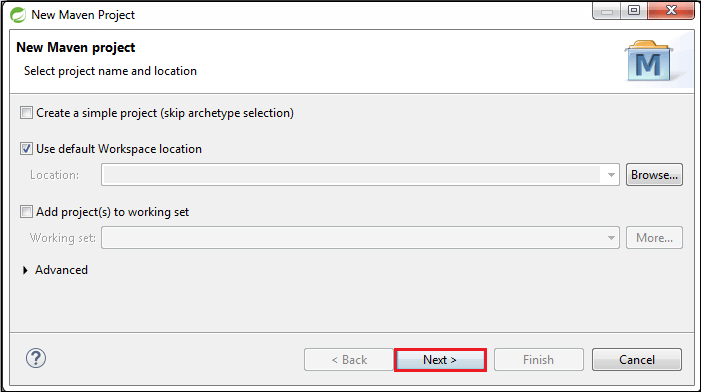
We can also use Spring Tool Suite to create a Spring project. In this section, we will create a **Maven Project** using **STS**.

**Step 1:** Open the Spring Tool Suite.

**Step 2:** Click on the File menu -> New -> Maven Project

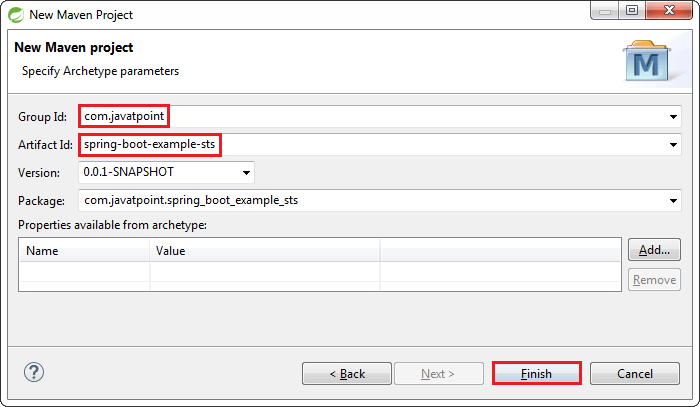


It shows the New Maven Project wizard. Click on the **Next** button.

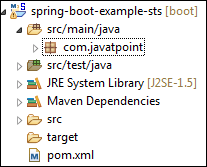


**Step 3:** Select the **maven-archetype-quickstart** and click on the **Next** button.

**Step 4:** Provide the **Group Id** and **Artifact Id**. We have provided Group Id **com.javatpoint** and Artifact Id **spring-boot-example-sts**. Now click on the **Finish** button.



When we click on the Finish button, it creates the project directory, as shown in the following image.



**Step 5:** Open the **App.java** file. We found the following code that is by default.

**App.java**

1. **package** com.javatpoint;
2. **public** **class** App
3. {
4. **public** **static** **void** main( String[] args )
5. {
6. System.out.println( "Hello World!" );
7. }
8. }

The Maven project has a **pom.xml** file which contains the following default configuration.

**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/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.javatpoint**</groupId>**
5. **<artifactId>**spring-boot-example-sts**</artifactId>**
6. **<version>**0.0.1-SNAPSHOT**</version>**
7. **<packaging>**jar**</packaging>**
8. **<name>**spring-boot-example-sts**</name>**
9. **<url>**http://maven.apache.org**</url>**
10. **<properties>**
11. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
12. **</properties>**
13. **<dependencies>**
14. **<dependency>**
15. **<groupId>**junit**</groupId>**
16. **<artifactId>**junit**</artifactId>**
17. **<version>**3.8.1**</version>**
18. **<scope>**test**</scope>**
19. **</dependency>**
20. **</dependencies>**
21. **</project>**

**Step 6:** Add **Java version** inside the **<properties>** tag.

1. <java.version>1.8</java.version>

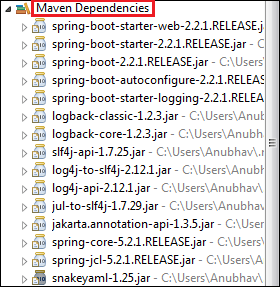
**Step 7:** In order to make a Spring Boot Project, we need to configure it. So, we are adding **spring boot starter parent** dependency in **pom.xml** file. Parent is used to declare that our project is a child to this parent project.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-parent**</artifactId>**
4. **<version>**2.2.1.RELEASE**</version>**
5. **<type>**pom**</type>**
6. **</dependency>**

**Step 8:** Add the **spring-boot-starter-web** dependency in **pom.xml** file.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **<version>**2.2.1.RELEASE**</version>**
5. **</dependency>**

#### **Note: When we add the dependencies in the pom file, it downloads the related jar file. We can see the downloaded jar files in the Maven Dependencies folder of the project directory.**



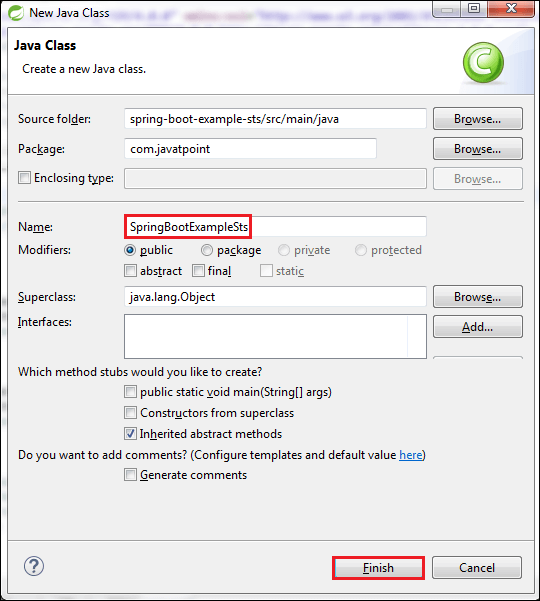
After adding all the dependencies, the pom.xml file looks like the following:

**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/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.javatpoint**</groupId>**
5. **<artifactId>**spring-boot-example-sts**</artifactId>**
6. **<version>**0.0.1-SNAPSHOT**</version>**
7. **<packaging>**jar**</packaging>**
8. **<name>**spring-boot-example-sts**</name>**
9. **<url>**http://maven.apache.org**</url>**
10. **<properties>**
11. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
12. **<java.version>**1.8**</java.version>**
13. **</properties>**
14. **<dependencies>**
15. **<dependency>**
16. **<groupId>**org.springframework.boot**</groupId>**
17. **<artifactId>**spring-boot-starter-parent**</artifactId>**
18. **<version>**2.2.1.RELEASE**</version>**
19. **<type>**pom**</type>**
20. **</dependency>**
21. **<dependency>**
22. **<groupId>**org.springframework.boot**</groupId>**
23. **<artifactId>**spring-boot-starter-web**</artifactId>**
24. **<version>**2.2.1.RELEASE**</version>**
25. **</dependency>**
26. **<dependency>**
27. **<groupId>**junit**</groupId>**
28. **<artifactId>**junit**</artifactId>**
29. **<version>**3.8.1**</version>**
30. **<scope>**test**</scope>**
31. **</dependency>**
32. **</dependencies>**
33. **</project>**

**Step 9:** Create a class with the name **SpringBootExampleSts** in the package **com.javatpoint**.

Right-click on the package name -> New -> Class -> provide the class name -> Finish



**Step 10:** After creating the class file, call the static method **run()** of the SpringApplication class. In the following code, we are calling the run() method and passing the class name as an argument.

1. SpringApplication.run(SpringBootExampleSts.**class**, args);

**Step 11:** Annotate the class by adding an annotation **@SpringBootApplication**.

**@SpringBootApplication**

A single @SpringBootApplication annotation is used to enable the following annotations:

* **@EnableAutoConfiguration:** It enables the Spring Boot auto-configuration mechanism.
* **@ComponentScan:** It scans the package where the application is located.
* **@Configuration:** It allows us to register extra beans in the context or import additional configuration classes.

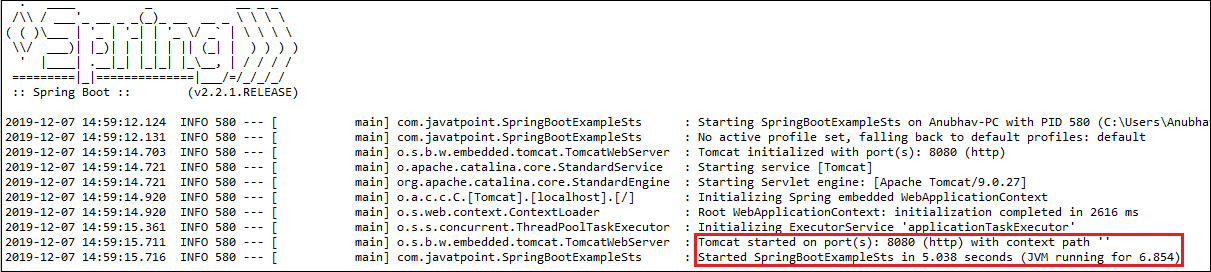
**SpringBootApplicationSts.java**

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

**Step:** Run the file **SpringBootExampleSts.java**, as Java Application. It displays the following in the console.

The line **Started SpringBootExampleSts in 5.038 seconds (JVM running for 6.854)** in the console shows that the application is up and running.

# Spring Boot Annotations

Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide **supp**l**emental** information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

In this section, we are going to discuss some important **Spring Boot Annotation** that we will use later in this tutorial.

## Core Spring Framework Annotations

**@Required:** It applies to the **bean** setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception **BeanInitilizationException**.

**Example**

1. **public** **class** Machine
2. {
3. **private** Integer cost;
4. @Required
5. **public** **void** setCost(Integer cost)
6. {
7. **this**.cost = cost;
8. }
9. **public** Integer getCost()
10. {
11. **return** cost;
12. }
13. }

**@Autowired:** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

**Example**

1. @Component
2. **public** **class** Customer
3. {
4. **private** Person person;
5. @Autowired
6. **public** Customer(Person person)
7. {
8. **this**.person=person;
9. }
10. }

**@Configuration:** It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

**Example**

1. @Configuration
2. **public** **class** Vehicle
3. {
4. @BeanVehicle engine()
5. {
6. **return** **new** Vehicle();
7. }
8. }

**@ComponentScan:** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

**Example**

1. @ComponentScan(basePackages = "com.javatpoint")
2. @Configuration
3. **public** **class** ScanComponent
4. {
5. // ...
6. }

**@Bean:** It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

**Example**

1. @Bean
2. **public** BeanExample beanExample()
3. {
4. **return** **new** BeanExample ();
5. }

## Spring Framework Stereotype Annotations

**@Component:** It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with **@Component** is found during the classpath. The Spring Framework pick it up and configure it in the application context as a **Spring Bean**.

**Example**

1. @Component
2. **public** **class** Student
3. {
4. .......
5. }

**@Controller:** The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

**Example**

1. @Controller
2. @RequestMapping("books")
3. **public** **class** BooksController
4. {
5. @RequestMapping(value = "/{name}", method = RequestMethod.GET)
6. **public** Employee getBooksByName()
7. {
8. **return** booksTemplate;
9. }
10. }

**@Service:** It is also used at class level. It tells the Spring that class contains the **business logic**.

**Example**

1. **package** com.javatpoint;
2. @Service
3. **public** **class** TestService
4. {
5. **public** **void** service1()
6. {
7. //business code
8. }
9. }

**@Repository:** It is a class-level annotation. The repository is a **DAOs** (Data Access Object) that access the database directly. The repository does all the operations related to the database.

1. **package** com.javatpoint;
2. @Repository
3. **public** **class** TestRepository
4. {
5. **public** **void** delete()
6. {
7. //persistence code
8. }
9. }

## Spring Boot Annotations

* **@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. **@SpringBootApplication**.
* **@SpringBootApplication:** It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan,** and **@Configuration**.

### **Spring MVC and REST Annotations**

* **@RequestMapping:** It is used to map the **web requests**. It has many optional elements like **consumes, header, method, name, params, path, produces**, and **value**. We use it with the class as well as the method.

**Example**

1. @Controller
2. **public** **class** BooksController
3. {
4. @RequestMapping("/computer-science/books")
5. **public** String getAllBooks(Model model)
6. {
7. //application code
8. **return** "bookList";
9. }

* **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**
* **@PostMapping:** It maps the **HTTP POST**requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
* **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**
* **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web service endpoint that **deletes**a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**
* **@PatchMapping:** It maps the **HTTP PATCH**requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**
* **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.
* **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.
* **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.
* **@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.
* **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.**For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
* **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
* **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

# Spring Boot Dependency Management

Spring Boot manages dependencies and configuration automatically. Each release of Spring Boot provides a list of dependencies that it supports. The list of dependencies is available as a part of the **Bills of Materials** (spring-boot-dependencies) that can be used with **Maven**. So, we need not to specify the version of the dependencies in our configuration. Spring Boot manages itself. Spring Boot upgrades all dependencies automatically in a consistent way when we update the Spring Boot version.

## Advantages of Dependency Management

* It provides the centralization of dependency information by specifying the Spring Boot version in one place. It helps when we switch from one version to another.
* It avoids mismatch of different versions of Spring Boot libraries.
* We only need to write a library name with specifying the version. It is helpful in multi-module projects.

#### **Note: Spring Boot also allows overriding of dependencies version, if required.**

## Maven Dependency Management System

The Maven project inherits the following features from **spring-boot-starter-parent:**

* The default **Java compiler version**
* **UTF-8** source encoding
* It inherits a **Dependency Section** from the spring-boot-dependency-pom. It manages the version of common dependencies. It ignores the **<version>** tag for that dependencies.
* Dependencies, inherited from the spring-boot-dependencies POM
* Sensible **resource filtering**
* Sensible **plugin configuration**

### **Inheriting Starter Parent**

The following **spring-boot-starter-parent** inherits automatically when we configure the project.

1. <parent>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-parent</artifactId>
4. <version>2.2.2.BUILD-SNAPSHOT</version>      <!-- lookup parent from repository -->
5. <relativePath/>
6. </parent>

#### **Note: In the above dependency, we have specified only the Spring Boot version. If we want to add additional starters, simply remove the <version> tag. Similarly, we can also override the individual dependency by overriding a property in our project.**

For example, if we want to add another dependency with the same artifact that we have injected already, inject that dependency again inside the **<properties>** tag to override the previous one.

### **Changing the Java version**

We can also change the Java version by using the **<java.version>** tag.

1. <properties>
2. <java.version>1.8</java.version>
3. </properties>

### **Adding Spring Boot Maven Plugin**

We can also **add Maven plugin** in our **pom.xml** file. It wraps the project into an executable **jar** file.

1. <build>
2. <plugins>
3. <plugin>
4. <groupId>org.springframework.boot</groupId>
5. <artifactId>spring-boot-maven-plugin</artifactId>
6. </plugin>
7. </plugins>
8. </build>

### **Spring Boot without Parent POM**

If we don't want to use **spring-boot starter-parent** dependency, but still want to take the advantage of the dependency management, we can use **<scope>** tag, as follows:

#### **Note: It does not maintain the plugin management.**

1. <dependencyManagement>
2. <dependencies>
3. <dependency><!-- Import dependency management from Spring Boot -->
4. <groupId>org.springframework.boot</groupId>
5. <artifactId>spring-boot-dependencies</artifactId>
6. <version>2.2.2.RELEASE</version>
7. <type>pom</type>
8. <scope>**import**</scope>
9. </dependency>
10. </dependencies>
11. </dependencyManagement>

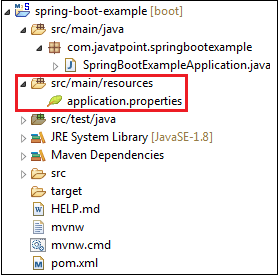
The above dependency does not allow overriding. To achieve the overriding, we need to add an entry inside the **<dependencyManagement>**tag of our project before the spring-boot-dependencies entry.

For example, to upgrade another **spring-data-releasetrain,**add the following dependency in the pom.xml file.

1. <dependencyManagement>
2. <dependencies>
3. <!--Override Spring Data release train-->
4. <dependency>
5. <groupId>org.springframework.data</groupId>
6. <artifactId>spring-data-releasetrain</artifactId>
7. <version>Fowler-SR2</version>
8. <type>pom</type>
9. <scope>**import**</scope>
10. </dependency>
11. <dependency>
12. <groupId>org.springframework.boot</groupId>
13. <artifactId>spring-boot-dependencies</artifactId>
14. <version>2.2.2.RELEASE</version>
15. <type>pom</type>
16. <scope>**import**</scope>
17. </dependency>
18. </dependencies>
19. </dependencyManagement>

# Spring Boot Application Properties

Spring Boot Framework comes with a built-in mechanism for application configuration using a file called **application.properties**. It is located inside the **src/main/resources** folder, as shown in the following figure.



Spring Boot provides various properties that can be configured in the **application.properties**file. The properties have default values. We can set a property(s) for the Spring Boot application. Spring Boot also allows us to define our own property if required.

The application.properties file allows us to run an application in a **different environment.**In short, we can use the application.properties file to:

* Configure the Spring Boot framework
* define our application custom configuration properties

### **Example of application.properties**

1. #configuring application name
2. spring.application.name = demoApplication
3. #configuring port
4. server.port = 8081

In the above example, we have configured the **application name** and **port**. The port 8081 denotes that the application runs on port **8081**.

#### **Note: The lines started with # are comments.**

**YAML Properties File**

Spring Boot provides another file to configure the properties is called **yml** file. The Yaml file works because the**Snake YAML** jar is present in the classpath. Instead of using the application.properties file, we can also use the application.yml file, but the **Yml** file should be present in the classpath.

**Example of application.yml**

1. spring:
2. application:
3. name: demoApplication
4. server:
5. port: 8081

In the above example, we have configured the **application name** and **port**. The port 8081 denotes that the application runs on port **8081**.

## Spring Boot Property Categories

There are **sixteen** categories of Spring Boot Property are as follows:

1. Core Properties
2. Cache Properties
3. Mail Properties
4. JSON Properties
5. Data Properties
6. Transaction Properties
7. Data Migration Properties
8. Integration Properties
9. Web Properties
10. Templating Properties
11. Server Properties
12. Security Properties
13. RSocket Properties
14. Actuator Properties
15. DevTools Properties
16. Testing Properties

## Application Properties Table

The following tables provide a list of common Spring Boot properties:

|  |  |  |
| --- | --- | --- |
| **Property** | **Default Values** | **Description** |
| Debug | False | It enables debug logs. |
| spring.application.name |  | It is used to set the application name. |
| spring.application.admin.enabled | False | It is used to enable admin features of the application. |
| spring.config.name | Application | It is used to set config file name. |
| spring.config.location |  | It is used to config the file name. |
| server.port | 8080 | Configures the HTTP server port |
| server.servlet.context-path |  | It configures the context path of the application. |
| logging.file.path |  | It configures the location of the log file. |
| spring.banner.charset | UTF-8 | Banner file encoding. |
| spring.banner.location | classpath:banner.txt | It is used to set banner file location. |
| logging.file |  | It is used to set log file name. For example, data.log. |
| spring.application.index |  | It is used to set application index. |
| spring.application.name |  | It is used to set the application name. |
| spring.application.admin.enabled | False | It is used to enable admin features for the application. |
| spring.config.location |  | It is used to config the file locations. |
| spring.config.name | Application | It is used to set config the file name. |
| spring.mail.default-encoding | UTF-8 | It is used to set default MimeMessage encoding. |
| spring.mail.host |  | It is used to set SMTP server host. For example, smtp.example.com. |
| spring.mail.password |  | It is used to set login password of the SMTP server. |
| spring.mail.port |  | It is used to set SMTP server port. |
| spring.mail.test-connection | False | It is used to test that the mail server is available on startup. |
| spring.mail.username |  | It is used to set login user of the SMTP server. |
| spring.main.sources |  | It is used to set sources for the application. |
| server.address |  | It is used to set network address to which the server should bind to. |
| server.connection-timeout |  | It is used to set time in milliseconds that connectors will wait for another HTTP request before closing the connection. |
| server.context-path |  | It is used to set context path of the application. |
| server.port | 8080 | It is used to set HTTP port. |
| server.server-header |  | It is used for the Server response header (no header is sent if empty) |
| server.servlet-path | / | It is used to set path of the main dispatcher servlet |
| server.ssl.enabled |  | It is used to enable SSL support. |
| spring.http.multipart.enabled | True | It is used to enable support of multi-part uploads. |
| spring.servlet.multipart.max-file-size | 1MB | It is used to set max file size. |
| spring.mvc.async.request-timeout |  | It is used to set time in milliseconds. |
| spring.mvc.date-format |  | It is used to set date format. For example, dd/MM/yyyy. |
| spring.mvc.locale |  | It is used to set locale for the application. |
| spring.social.facebook.app-id |  | It is used to set application's Facebook App ID. |
| spring.social.linkedin.app-id |  | It is used to set application's LinkedIn App ID. |
| spring.social.twitter.app-id |  | It is used to set application's Twitter App ID. |
| security.basic.authorize-mode | Role | It is used to set security authorize mode to apply. |
| security.basic.enabled | True | It is used to enable basic authentication. |
| Spring.test.database.replace | Any | Type of existing DataSource to replace. |
| Spring.test.mockmvc.print | Default | MVC Print option |
| spring.freemaker.content-type | text/html | Content Type value |
| server.server-header |  | Value to use for the server response header. |
| spring.security.filter.dispatcher-type | async, error, request | Security filter chain dispatcher types. |
| spring.security.filter.order | -100 | Security filter chain order. |
| spring.security.oauth2.client.registration.\* |  | OAuth client registrations. |
| spring.security.oauth2.client.provider.\* |  | OAuth provider details. |

# Spring Boot Starters

**Spring Boot** provides a number of **starters** that allow us to add jars in the classpath. Spring Boot built-in**starters** make development easier and rapid.**Spring Boot Starters** are the **dependency descriptors**.

In the Spring Boot Framework, all the starters follow a similar naming pattern: **spring-boot-starter-\***, where **\***denotes a particular type of application. For example, if we want to use Spring and JPA for database access, we need to include the **spring-boot-starter-data-jpa** dependency in our **pom.xml** file of the project.

## Third-Party Starters

We can also include **third party starters** in our project. But we do not use **spring-boot-starter** for including third party dependency. The spring-boot-starter is reserved for official Spring Boot artifacts. The third-party starter starts with the name of the project. For example, the third-party project name is **abc,**then the dependency name will be**abc-spring-boot-starter.**

The Spring Boot Framework provides the following application starters under the **org.springframework.boot** group.

|  |  |
| --- | --- |
| **Name** | **Description** |
| spring-boot-starter-thymeleaf | It is used to build MVC web applications using Thymeleaf views. |
| spring-boot-starter-data-couchbase | It is used for the Couchbase document-oriented database and Spring Data Couchbase. |
| spring-boot-starter-artemis | It is used for JMS messaging using Apache Artemis. |
| spring-boot-starter-web-services | It is used for Spring Web Services. |
| spring-boot-starter-mail | It is used to support Java Mail and Spring Framework's email sending. |
| spring-boot-starter-data-redis | It is used for Redis key-value data store with Spring Data Redis and the Jedis client. |
| spring-boot-starter-web | It is used for building the web application, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container. |
| spring-boot-starter-data-gemfire | It is used to GemFire distributed data store and Spring Data GemFire. |
| spring-boot-starter-activemq | It is used in JMS messaging using Apache ActiveMQ. |
| spring-boot-starter-data-elasticsearch | It is used in Elasticsearch search and analytics engine and Spring Data Elasticsearch. |
| spring-boot-starter-integration | It is used for Spring Integration. |
| spring-boot-starter-test | It is used to test Spring Boot applications with libraries, including JUnit, Hamcrest, and Mockito. |
| spring-boot-starter-jdbc | It is used for JDBC with the Tomcat JDBC connection pool. |
| spring-boot-starter-mobile | It is used for building web applications using Spring Mobile. |
| spring-boot-starter-validation | It is used for Java Bean Validation with Hibernate Validator. |
| spring-boot-starter-hateoas | It is used to build a hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS. |
| spring-boot-starter-jersey | It is used to build RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web. |
| spring-boot-starter-data-neo4j | It is used for the Neo4j graph database and Spring Data Neo4j. |
| spring-boot-starter-data-ldap | It is used for Spring Data LDAP. |
| spring-boot-starter-websocket | It is used for building the WebSocket applications. It uses Spring Framework's WebSocket support. |
| spring-boot-starter-aop | It is used for aspect-oriented programming with Spring AOP and AspectJ. |
| spring-boot-starter-amqp | It is used for Spring AMQP and Rabbit MQ. |
| spring-boot-starter-data-cassandra | It is used for Cassandra distributed database and Spring Data Cassandra. |
| spring-boot-starter-social-facebook | It is used for Spring Social Facebook. |
| spring-boot-starter-jta-atomikos | It is used for JTA transactions using Atomikos. |
| spring-boot-starter-security | It is used for Spring Security. |
| spring-boot-starter-mustache | It is used for building MVC web applications using Mustache views. |
| spring-boot-starter-data-jpa | It is used for Spring Data JPA with Hibernate. |
| spring-boot-starter | It is used for core starter, including auto-configuration support, logging, and YAML. |
| spring-boot-starter-groovy-templates | It is used for building MVC web applications using Groovy Template views. |
| spring-boot-starter-freemarker | It is used for building MVC web applications using FreeMarker views. |
| spring-boot-starter-batch | It is used for Spring Batch. |
| spring-boot-starter-social-linkedin | It is used for Spring Social LinkedIn. |
| spring-boot-starter-cache | It is used for Spring Framework's caching support. |
| spring-boot-starter-data-solr | It is used for the Apache Solr search platform with Spring Data Solr. |
| spring-boot-starter-data-mongodb | It is used for MongoDB document-oriented database and Spring Data MongoDB. |
| spring-boot-starter-jooq | It is used for jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc. |
| spring-boot-starter-jta-narayana | It is used for Spring Boot Narayana JTA Starter. |
| spring-boot-starter-cloud-connectors | It is used for Spring Cloud Connectors that simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku. |
| spring-boot-starter-jta-bitronix | It is used for JTA transactions using Bitronix. |
| spring-boot-starter-social-twitter | It is used for Spring Social Twitter. |
| spring-boot-starter-data-rest | It is used for exposing Spring Data repositories over REST using Spring Data REST. |

## Spring Boot Production Starters

|  |  |
| --- | --- |
| **Name** | **Description** |
| spring-boot-starter-actuator | It is used for Spring Boot's Actuator that provides production-ready features to help you monitor and manage your application. |
| spring-boot-starter-remote-shell | It is used for the CRaSH remote shell to monitor and manage your application over SSH. Deprecated since 1.5. |

## Spring Boot Technical Starters

|  |  |
| --- | --- |
| **Name** | **Description** |
| spring-boot-starter-undertow | It is used for Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat. |
| spring-boot-starter-jetty | It is used for Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat. |
| spring-boot-starter-logging | It is used for logging using Logback. Default logging starter. |
| spring-boot-starter-tomcat | It is used for Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web. |
| spring-boot-starter-log4j2 | It is used for Log4j2 for logging. An alternative to spring-boot-starter-logging. |

Spring Boot Starter Parent

Spring Boot Starter Parent

The spring-boot-starter-parent is a project starter. It provides default configurations for our applications. It is used internally by all dependencies. All Spring Boot projects use spring-boot-starter-parent as a parent in pom.xml file.

1. <parent>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-parent</artifactId>
4. <version>1.4.0.RELEASE</version>
5. </parent>

Parent Poms allow us to manage the following things for multiple child projects and modules:

* **Configuration:** It allows us to maintain consistency of Java Version and other related properties.
* **Dependency Management:** It controls the versions of dependencies to avoid conflict.
* Source encoding
* Default Java Version
* Resource filtering
* It also controls the default plugin configuration.

The spring-boot-starter-parent inherits dependency management from spring-boot-dependencies. We only need to specify the Spring Boot version number. If there is a requirement of the additional starter, we can safely omit the version number.

Spring Boot Starter Parent Internal

Spring Boot Starter Parent defines spring-boot-dependencies as a parent pom. It inherits dependency management from spring-boot-dependencies.

1. <parent>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-dependencies</artifactId>
4. <version>1.6.0.RELEASE</version>
5. <relativePath>../../spring-boot-dependencies</relativePath>
6. </parent>

**Default Parent Pom**

1. <properties>
2. <java.version>1.8</java.version>
3. <resource.delimiter>@</resource.delimiter>
4. <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
5. <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
6. <maven.compiler.source>${java.version}</maven.compiler.source>
7. <maven.compiler.target>${java.version}</maven.compiler.target>
8. </properties>

The properties section defines the application default values. The default Java version is 1.8. We can also override Java version by specifying a property **<java.version>1.8</java.version>** in the project pom. The parent pom also contains the few other settings related to encoding and source. The Spring Boot framework uses these defaults in case, if we have not defined in the application.properties file.

**Plugin Management**

The **spring-boot-starter-parent** specifies the default configuration for a host of plugins including maven-failsafe-plugin, maven-jar-plugin and maven-surefire-plugin.

1. <plugin>
2. <groupId>org.apache.maven.plugins</groupId>
3. <artifactId>maven-failsafe-plugin</artifactId>
4. <executions>
5. <execution>
6. <goals>
7. <goal>integration-test</goal>
8. <goal>verify</goal>
9. </goals>
10. </execution>
11. </executions>
12. </plugin>
13. <plugin>
14. <groupId>org.apache.maven.plugins</groupId>
15. <artifactId>maven-jar-plugin</artifactId>
16. <configuration>
17. <archive>
18. <manifest>
19. <mainClass>${start-**class**}</mainClass> <addDefaultImplementationEntries>**true**</addDefaultImplementationEntries>
20. </manifest>
21. </archive>
22. </configuration>
23. </plugin>
24. <plugin>
25. <groupId>org.apache.maven.plugins</groupId>
26. <artifactId>maven-surefire-plugin</artifactId>
27. <configuration>
28. <includes>
29. <include>\*\*/\*Tests.java</include>
30. <include>\*\*/\*Test.java</include>
31. </includes>
32. <excludes>
33. <exclude>\*\*/Abstract\*.java</exclude>
34. </excludes>
35. </configuration>
36. </plugin>

**Spring Boot Dependencies**

The spring-boot-starter-parent dependency inherit from the spring-boot-dependencies, it shares all these characteristics as well. Hence the Spring Boot manages the list of the dependencies as the part of the dependency management.

1. <properties>
2. <activemq.version>5.13.4</activemq.version>
3. ...
4. <ehcache.version>2.10.2.2.21</ehcache.version>
5. <ehcache3.version>3.1.1</ehcache3.version>
6. ...
7. <h2.version>1.4.192</h2.version>
8. <hamcrest.version>1.3</hamcrest.version>
9. <hazelcast.version>3.6.4</hazelcast.version>
10. <hibernate.version>5.0.9.Final</hibernate.version>
11. <hibernate-validator.version>5.2.4.Final</hibernate-validator.version>
12. <hikaricp.version>2.4.7</hikaricp.version>
13. <hikaricp-java6.version>2.3.13</hikaricp-java6.version>
14. <hornetq.version>2.4.7.Final</hornetq.version>
15. <hsqldb.version>2.3.3</hsqldb.version>
16. <htmlunit.version>2.21</htmlunit.version>
17. <httpasyncclient.version>4.1.2</httpasyncclient.version>
18. <httpclient.version>4.5.2</httpclient.version>
19. <httpcore.version>4.4.5</httpcore.version>
20. <infinispan.version>8.2.2.Final</infinispan.version>
21. <jackson.version>2.8.1</jackson.version>
22. ....
23. <jersey.version>2.23.1</jersey.version>
24. <jest.version>2.0.3</jest.version>
25. <jetty.version>9.3.11.v20160721</jetty.version>
26. <jetty-jsp.version>2.2.0.v201112011158</jetty-jsp.version>
27. <spring-security.version>4.1.1.RELEASE</spring-security.version>
28. <tomcat.version>8.5.4</tomcat.version>
29. <undertow.version>1.3.23.Final</undertow.version>
30. <velocity.version>1.7</velocity.version>
31. <velocity-tools.version>2.0</velocity-tools.version>
32. <webjars-hal-browser.version>9f96c74</webjars-hal-browser.version>
33. <webjars-locator.version>0.32</webjars-locator.version>
34. <wsdl4j.version>1.6.3</wsdl4j.version>
35. <xml-apis.version>1.4.01</xml-apis.version>
36. </properties>
37. <prerequisites>
38. <maven>3.2.1</maven>
39. </prerequisites>

Spring Boot Starter without Parent

In some cases, we need not to inherit spring-boot-starter-parent in the pom.xml file. To handle such use cases, Spring Boot provides the flexibility to still use the dependency management without inheriting the spring-boot-starter-parent.

1. <dependencyManagement>
2. <dependencies>
3. <dependency>
4. <!-- Import dependency management from Spring Boot -->
5. <groupId>org.springframework.boot</groupId>
6. <artifactId>spring-boot-dependencies</artifactId>
7. <version>2.1.1.RELEASE</version>
8. <type>pom</type>
9. <scope>**import**</scope>
10. </dependency>
11. </dependencies>
12. </dependencyManagement>

In the above code, we can see that we have used **<scope>** tag for this. It is useful when we want to use different version for a certain dependency.

# Spring Boot Starter Web

There are two important features of spring-boot-starter-web:

* It is compatible for web development
* Auto configuration

If we want to develop a web application, we need to add the following dependency in pom.xml file:

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **<version>**2.2.2.RELEASE**</version>**
5. **</dependency>**

Starter of Spring web uses Spring MVC, REST and Tomcat as a default embedded server. The single spring-boot-starter-web dependency transitively pulls in all dependencies related to web development. It also reduces the build dependency count. The spring-boot-starter-web transitively depends on the following:

* org.springframework.boot:spring-boot-starter
* org.springframework.boot:spring-boot-starter-tomcat
* org.springframework.boot:spring-boot-starter-validation
* com.fasterxml.jackson.core:jackson-databind
* org.springframework:spring-web
* org.springframework:spring-webmvc

By default, the spring-boot-starter-web contains the following tomcat server dependency:

Prime Ministers of India | List of Prime Minister of India (1947-2020)

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-tomcat**</artifactId>**
4. **<version>**2.0.0.RELEASE**</version>**
5. **<scope>**compile**</scope>**
6. **</dependency>**

The spring-boot-starter-web auto-configures the following things that are required for the web development:

* Dispatcher Servlet
* Error Page
* Web JARs for managing the static dependencies
* Embedded servlet container

## Spring Boot Embedded Web Server

Each Spring Boot application includes an embedded server. Embedded server is embedded as a part of deployable application. The advantage of embedded server is, we do not require pre-installed server in the environment. With Spring Boot, default embedded server is **Tomcat**. Spring Boot also supports another two embedded servers:

* **Jetty Server**
* **Undertow Server**

## Using another embedded web server

For **servlet stack** applications, the **spring-boot-starter-web** includes **Tomcat** by including **spring-boot-starter-tomcat**, but we can use **spring-boot-starter-jetty** or **spring-boot-starter-undertow** instead.

For **reactive stack** applications, the **spring-boot-starter-webflux** includes **Reactor Netty**by including **spring-boot-starter-reactor-netty**, but we can use **spring-boot-starter-tomcat, spring-boot-starter-jetty,** or **spring-boot-starter-undertow**instead.

### **Jetty Server**

The Spring Boot also supports an embedded server called **Jetty Server**. It is an HTTP server and Servlet container that has the capability of serving static and dynamic content. It is used when machine to machine communication is required.

If we want to add the Jetty server in the application, we need to add the **spring-boot-starter-jetty** dependency in our pom.xml file.

**Remember:** While using Jetty server in the application, make sure that the default Tomcat server is **excluded** from the **spring-boot-starter-web.** It avoids the conflict between servers.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **<exclusions>**
5. **<exclusion>**
6. **<groupId>**org.springframework.boot**</groupId>**
7. **<artifactId>**spring-boot-starter-tomcat**</artifactId>**
8. **</exclusion>**
9. **</exclusions>**
10. **</dependency>**
11. **<dependency>**
12. **<groupId>**org.springframework.boot**</groupId>**
13. **<artifactId>**spring-boot-starter-jetty**</artifactId>**
14. **</dependency>**

We can also customize the behavior of the Jetty server by using the **application.properties** file.

### **Undertow Server**

Spring Boot provides another server called **Undertow**. It is also an embedded web server like Jetty. It is written in Java and manage and sponsored by JBoss. The main advantages of Undertow server are:

* Supports HTTP/2
* HTTP upgrade support
* Websocket Support
* Provides support for Servlet 4.0
* Flexible
* Embeddable

**Remember:** While using Undertow server in the application, make sure that the default Tomcat server is **excluded** from the **spring-boot-starter-web.** It avoids the conflict between servers.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **<exclusions>**
5. **<exclusion>**
6. **<groupId>**org.springframework.boot**</groupId>**
7. **<artifactId>**spring-boot-starter-tomcat**</artifactId>**
8. **</exclusion>**
9. **</exclusions>**
10. **</dependency>**
11. **<dependency>**
12. **<groupId>**org.springframework.boot**</groupId>**
13. **<artifactId>**spring-boot-starter-undertow**</artifactId>**
14. **</dependency>**

We can also customize the behavior of the Undertow server by using the **application.properties** file.

## spring-boot-starter-web vs. spring-boot-starter-tomcat

The spring-boot-starter-web contains the spring web dependencies that includes spring-boot-starter-tomcat. The spring-boot-starter-web contains the following:

* spring-boot-starter
* jackson
* spring-core
* spring-mvc
* spring-boot-starter-tomcat

While the **spring-boot-starter-tomcat** contains everything related to Tomcat server.

* core
* el
* logging
* websocket

The starter-tomcat has the following dependencies:

1. **<dependency>**
2. **<groupId>**org.apache.tomcat.embed**</groupId>**
3. **<artifactId>**tomcat-embed-core**</artifactId>**
4. **<version>**8.5.23**</version>**
5. **<scope>**compile**</scope>**
6. **</dependency>**
7. **<dependency>**
8. **<groupId>**org.apache.tomcat.embed**</groupId>**
9. **<artifactId>**tomcat-embed-el**</artifactId>**
10. **<version>**8.5.23**</version>**
11. **<scope>**compile**</scope>**
12. **</dependency>**
13. **<dependency>**
14. **<groupId>**org.apache.tomcat.embed**</groupId>**
15. **<artifactId>**tomcat-embed-websocket**</artifactId>**
16. **<version>**8.5.23**</version>**
17. **<scope>**compile**</scope>**
18. **</dependency>**

We can also use **spring-mvc** without using the embedded Tomcat server. If we want to do so, we need to exclude the Tomcat server by using the **<exclusion>** tag, as shown in the following code.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **<exclusions>**
5. **<exclusion>**
6. **<groupId>**org.springframework.boot**</groupId>**
7. **<artifactId>**spring-boot-starter-tomcat**</artifactId>**
8. **</exclusion>**
9. **</exclusions>**
10. **</dependency>**

# Spring Data JPA

Spring Data is a high-level Spring Source project. Its purpose is to unify and easy access to the different kinds of persistence stores, both relational database systems, and NoSQL data stores.

When we implement a new application, we should focus on the business logic instead of technical complexity and boilerplate code. That's why the Java Persistent API (JPA) specification and Spring Data JPA are extremely popular.

Spring Data JPA adds a layer on the top of JPA. It means, Spring Data JPA uses all features defined by JPA specification, especially the entity, association mappings, and JPA's query capabilities. Spring Data JPA adds its own features such as the no-code implementation of the repository pattern and the creation of database queries from the method name.

## Spring Data JPA

Spring Data JPA handles most of the complexity of JDBC-based database access and ORM (Object Relational Mapping). It reduces the boilerplate code required by JPA. It makes the implementation of your persistence layer easier and faster.

10 Sec

History of Java

Spring Data JPA aims to improve the implementation of data access layers by reducing the effort to the amount that is needed.

### **Spring Data JPA Features**

There are**three** main features of Spring Data JPA are as follows:

* **No-code repository:** It is the most popular persistence-related pattern. It enables us to implement our business code on a higher abstraction level.
* **Reduced boilerplate code:** It provides the default implementation for each method by its repository interfaces. It means that there is no longer need to implement read and write operations.
* **Generated Queries:** Another feature of Spring Data JPA is the **generation of database queries** based on the method name. If the query is not too complex, we need to define a method on our repository interface with the name that starts with **findBy**. After defining the method, Spring parses the method name and creates a query for it. For example:

1. public interface EmployeeRepository extends CrudRepository**<Employee**, Long**>**
2. {
3. Employee findByName(String name);
4. }

In the above example, we extend the **CrudRepository** that uses two generics: **Employee** and **Long**. The Employee is the **entity** that is to be managed, and **Long**is the data type of primary key

Spring internally generates a **JPQL** (Java Persistence Query Language) query based on the method name. The query is derived from the method signature. It sets the bind parameter value, execute the query, and returns the result.

There are some other features are as follows:

* It can integrate custom repository code.
* It is a powerful repository and custom object-mapping abstraction.
* It supports transparent auditing.
* It implements a domain base class that provides basic properties.
* It supports several modules such as Spring Data JPA, Spring Data MongoDB, Spring Data REST, Spring Data Cassandra, etc.

## Spring Data Repository

Spring Data JPA provides **three** repositories are as follows:

* **CrudRepository:** It offers standard **create, read, update,** and **delete** It contains method like **findOne(), findAll(), save(), delete(),**etc.
* **PagingAndSortingRepository:** It extends the **CrudRepository** and adds the findAll methods. It allows us to **sort** and **retrieve** the data in a paginated way.
* **JpaRepository:** It is a **JPA specific repository** It is defined in **Spring Data Jpa**. It extends the both repository CrudRepository and PagingAndSortingRepository. It adds the JPA-specific methods, like **flush()** to trigger a flush on the persistence context.

1. **<dependency>**
2. **<groupId>**org.springframework.data**</groupId>**
3. **<artifactId>**spring-data-jpa**</artifactId>**
4. **<version>**2.2.3.RELEASE**</version>**
5. **</dependency>**

## Spring Boot Starter Data JPA

Spring Boot provides **spring-boot-starter-data-jpa** dependency to connect Spring application with relational database efficiently. The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency (since Spring Boot version 1.5.3).

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-data-jpa**</artifactId>**
4. **<version>**2.2.2.RELEASE**</version>**
5. **</dependency>**

The databases are designed with tables/relations. Earlier approaches (JDBC) involved writing SQL queries.  In the JPA, we will store the data from objects into table and vice-versa. However, JPA evolved as a result of a different thought process.

Before JPA, ORM was the term more commonly used to refer to these frameworks. It is the reason Hibernate is called the ORM framework.

JPA allows us to map application classes to table in the database.

* **Entity Manager:** Once we define the mapping, it handles all the interactions with the database.
* **JPQL (Java Persistence Query Language):** It provides a way to write queries to execute searches against entities. It is different from the SQL queries. JPQL queries already understand the mapping that is defined between entities. We can add additional conditions if required.
* **Criteria API:** It defines a Java-based API to execute searches against the database.

### **Hibernate vs. JPA**

Hibernate is the implementation of JPA. It is the most popular ORM framework, while JPA is an API that defines the specification. Hibernate understands the mapping that we add between objects and tables. It ensures that data is retrieved/ stored from the database based on the mapping. It also provides additional features on the top of the JPA.

### **Spring Boot JPA Example**

In this example, we will use spring-boot-starter-data-jpa dependency to create a connection with the H2 database.

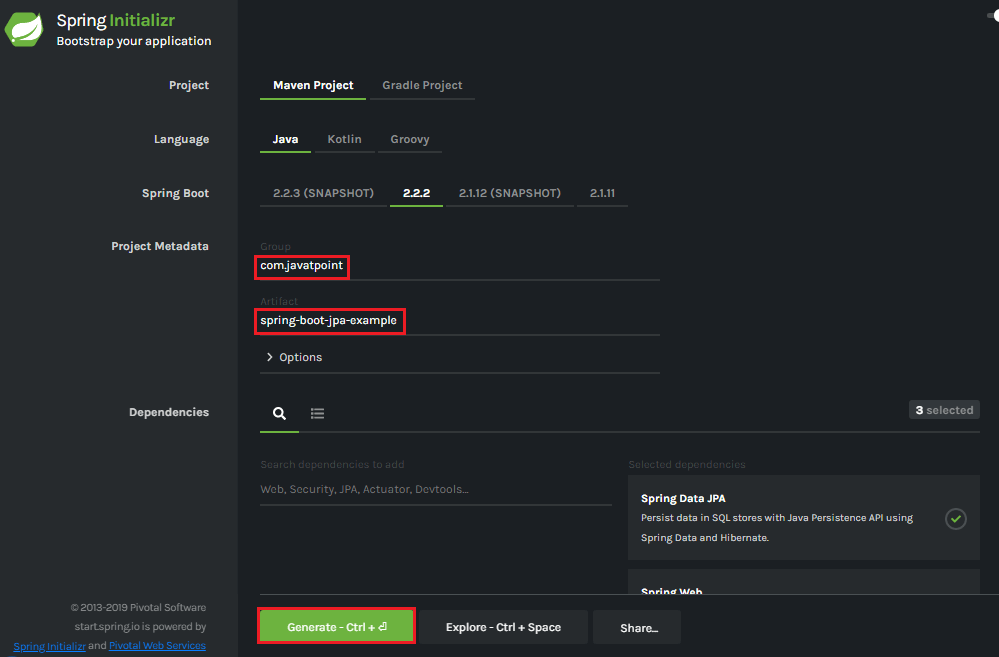
**Step 1:** Open spring Initializr <https://start.spring.io/>.

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint**.

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-jpa-example.**

**Step 4:** Add the dependencies: **Spring Web, Spring Data JPA,**and**H2 Database.**

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps the project in **Jar** file and downloads it to the local system.



**Step 6: Extract** the Jar file and paste it into the STS workspace.

**Step 7: Import** the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-jpa-example -> Finish

It takes some time to import.

**Step 8:**Create a package with the name **com.javatpoint.controller**in the folder **src/main/java**.

**Step 9:**Create a Controller class with the name **ControllerDemo** in the package **com.javatpoint.controller**.

**ControllerDemo.java**

1. package com.javatpoint.controller;
2. import org.springframework.stereotype.Controller;
3. import org.springframework.web.bind.annotation.RequestMapping;
4. @Controller
5. public class ControllerDemo
6. {
7. @RequestMapping("/")
8. public String home()
9. {
10. return "home.jsp";
11. }
12. }

**Step 10:**Create another package with the name **com.javatpoint.model**in the folder **src/main/java.**

**Step 11:**Create a class with the name **User**in the package com.javatpoint.model.

**User.java**

1. package com.javatpoint.model;
2. import javax.persistence.Entity;
3. import javax.persistence.Id;
4. import javax.persistence.Table;
5. @Entity
6. @Table(name="userdata")
7. public class User
8. {
9. @Id
10. private int id;
11. private String username;
12. public int getId()
13. {
14. return id;
15. }
16. public void setId(int id)
17. {
18. this.id = id;
19. }
20. public String getUname()
21. {
22. return username;
23. }
24. public void setUname(String username)
25. {
26. this.username = username;
27. }
28. @Override
29. public String toString()
30. {
31. return "User [id=" + id + ", uname=" + username + "]";
32. }
33. }

Now we need to Configure the H2 database.

**Step 12:** Open the **application.properties**file and configure the following things: **port, enable the H2 console, datasource,**and**URL.**

**application.properties**

1. server.port=8085
2. spring.h2.console.enabled=true
3. spring.datasource.plateform=h2
4. spring.datasource.url=jdbc:h2:mem:javatpoint

**Step 13:** Create a **SQL** file in the folder **src/main/resources.**

Right-click on the folder src/main/resources -> New -> File -> Provide the **File name** -> Finish

We have provided the file name **data.sql** and insert the following data into it.

**data.sql**

1. insert into userdata values(101,'Tom');
2. insert into userdata values(102,'Andrew');
3. insert into userdata values(103,'Tony');
4. insert into userdata values(104,'Bob');
5. insert into userdata values(105,'Sam');

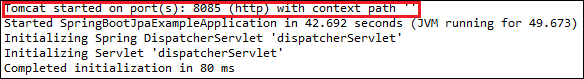
**Step 14:** Create a folder with the name **webapp** in the **src** folder.

**Step 15:** Create a JSP file with the name that we have returned in the **ControllerDemo**. In the ControllerDemo.java, we have returned **home.jsp**.

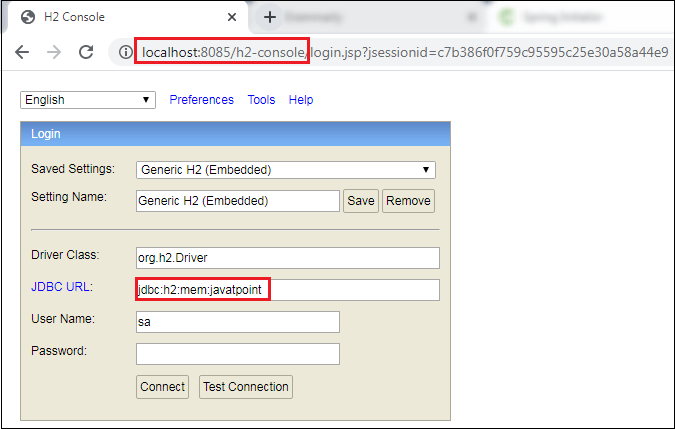
**home.jsp**

1. **<**%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2. pageEncoding="ISO-8859-1"%**>**
3. <!DOCTYPE html**>**
4. **<html>**
5. **<head>**
6. **<meta** charset="ISO-8859-1"**>**
7. **<title>**Insert title here**</title>**
8. **</head>**
9. **<body>**
10. **<form** action="addUser"**>**
11. ID :**<br** **/>**
12. **<input** type="text" name="t1"**><br** **/>**
13. User name :**<br** **/>**
14. **<input** type="text" name="t2"**><br** **/>**
15. **<input** type="submit" value="Add"**>**
16. **</form>**
17. **</body>**
18. **</html>**

**Step 16:** Run the **SpringBootJpaExampleApplication.java** file. We can see in the console that our application is successfully running on port **8085**.

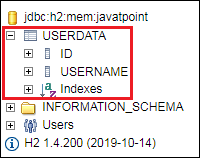


**Step 17:** Open the browser and invoke the URL http://localhost:8085/h2-console/. It shows the Driver Class, JDBC URL that we have configured in the **application.properties** file, and the default User Name sa.



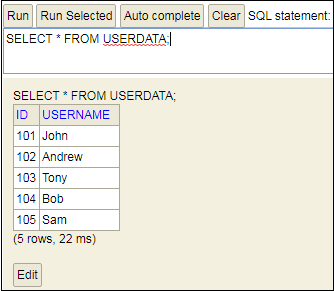
We can also test the connection by clicking on the **Test Connection** button. If the connection is successful, it shows a message Test Successful.

**Step 18:** Click on the **Connect** button. It shows the structure of the table userdata that we have defined in the **User.java**.



**Step 19:** Execute the following query to see the data that we have inserted in the **data.sql** file.

1. SELECT \* FROM USERDATA;



# Spring Boot Starter Actuator

## Spring Boot Actuator

**Spring Boot Actuator** is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use **HTTP** and **JMX** endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the S**pring Boot actuator.**

### **Spring Boot Actuator Features**

There are **three** main features of Spring Boot Actuator:

* **Endpoints**
* **Metrics**
* **Audit**

**Endpoint:** The actuator endpoints allows us to monitor and interact with the application. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application choose **HTTP**, where the Id of the endpoint, along with the prefix of **/actuator,**is mapped to a URL.

For example, the **/health** endpoint provides the basic health information of an application. The actuator, by default, mapped it to **/actuator/health**.

How to find Nth Highest Salary in SQL

**Metrics:** Spring Boot Actuator provides dimensional metrics by integrating with the**micrometer**. The micrometer is integrated into Spring Boot. It is the instrumentation library powering the delivery of application metrics from Spring. It provides vendor-neutral interfaces for **timers, gauges, counters, distribution summaries,** and **long task timers** with a dimensional data model.

**Audit:** Spring Boot provides a flexible audit framework that publishes events to an **AuditEventRepository.** It automatically publishes the authentication events if spring-security is in execution.

## Enabling Spring Boot Actuator

We can enable actuator by injecting the dependency **spring-boot-starter-actuator** in the pom.xml file.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-actuator**</artifactId>**
4. **<version>**2.2.2.RELEASE**</version>**
5. **</dependency>**

## Spring Boot Actuator Endpoints

The actuator endpoints allow us to monitor and interact with our Spring Boot application. Spring Boot includes number of built-in endpoints and we can also add custom endpoints in Spring Boot application.

The following table describes the widely used endpoints.

|  |  |  |
| --- | --- | --- |
| **Id** | **Usage** | **Default** |
| actuator | It provides a hypermedia-based **discovery page** for the other endpoints. It requires Spring HATEOAS to be on the classpath. | True |
| auditevents | It exposes audit events information for the current application. | True |
| autoconfig | It is used to display an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied. | True |
| beans | It is used to display a complete list of all the Spring beans in your application. | True |
| configprops | It is used to display a collated list of all @ConfigurationProperties. | True |
| dump | It is used to perform a thread dump. | True |
| env | It is used to expose properties from Spring's ConfigurableEnvironment. | True |
| flyway | It is used to show any Flyway database migrations that have been applied. | True |
| health | It is used to show application health information. | False |
| info | It is used to display arbitrary application info. | False |
| loggers | It is used to show and modify the configuration of loggers in the application. | True |
| liquibase | It is used to show any Liquibase database migrations that have been applied. | True |
| metrics | It is used to show metrics information for the current application. | True |
| mappings | It is used to display a collated list of all @RequestMapping paths. | True |
| shutdown | It is used to allow the application to be gracefully shutdown. | True |
| trace | It is used to display trace information. | True |

For Spring MVC, the following additional endpoints are used.

|  |  |  |
| --- | --- | --- |
| **Id** | **Description** | **Default** |
| docs | It is used to display documentation, including example requests and responses for the Actuator's endpoints. | False |
| heapdump | It is used to return a GZip compressed hprof heap dump file. | True |
| jolokia | It is used to expose JMX beans over HTTP (when Jolokia is on the classpath). | True |
| logfile | It is used to return the contents of the logfile. | True |
| prometheus | It is used to expose metrics in a format that can be scraped by a prometheus server. It requires a dependency on micrometer-registry- prometheus. | True |

## Spring Boot actuator properties

Spring Boot enables security for all actuator endpoints. It uses **form-based** authentication that provides **user Id** as the user and a randomly generated **password**. We can also access actuator-restricted endpoints by customizing basicauth security to the endpoints. We need to override this configuration by **management.security.roles** property. For example:

1. management.security.enabled=true
2. management.security.roles=ADMIN
3. security.basic.enabled=true
4. security.user.name=admin
5. security.user.passowrd=admin

### **Spring Boot Actuator Example**

Let's understand the concept of the actuator through an example.

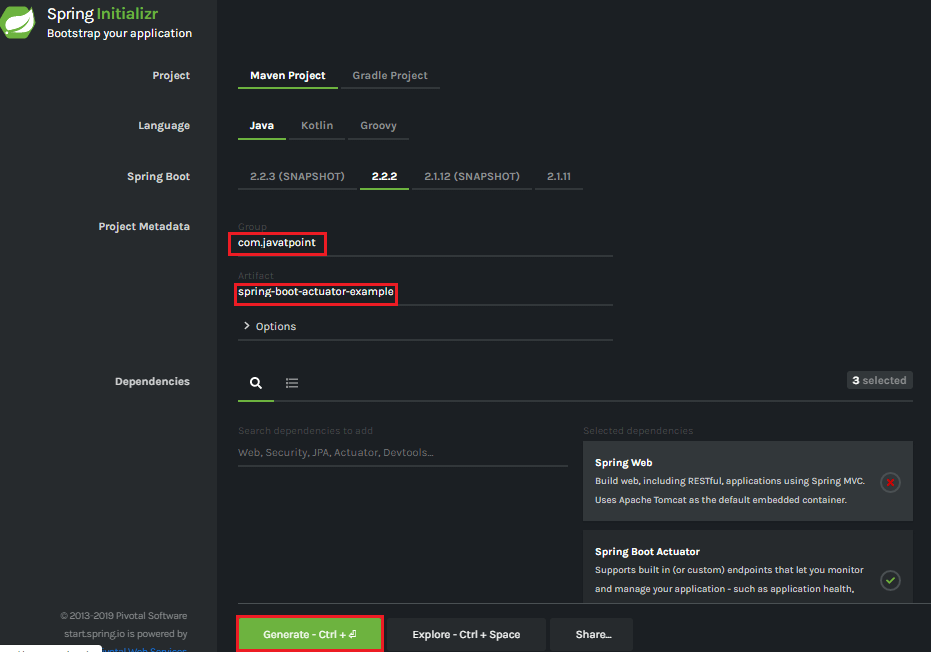
**Step 1:** Open Spring Initializr <https://start.spring.io/> and create a **Maven** project.

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided the **spring-boot-actuator-example.**

**Step 4:** Add the following dependencies: **Spring Web, Spring Boot Starter Actuator,** and **Spring Data Rest HAL Browser**.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications related to the project into a **Jar** file and downloads it to our local system.

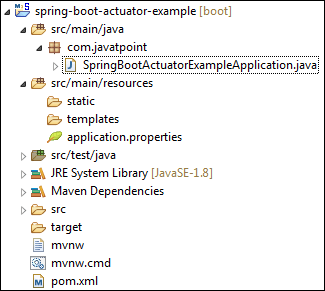


**Step 6:** Extract the Jar file and paste it into the STS workspace.

**Step 7:** Import the project folder.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-actuator-example -> Finish

It takes some time to import. After importing the project, we can see the project directory in the package explorer section.



**Step 8:** Create a Controller class. We have created the controller class with the name DemoRestController.

**DemoRestController.java**

1. package com.javatpoint;
2. import org.springframework.web.bind.annotation.GetMapping;
3. import org.springframework.web.bind.annotation.RestController;
4. @RestController
5. public class DemoRestController
6. {
7. @GetMapping("/hello")
8. public String hello()
9. {
10. return "Hello User!";
11. }
12. }

**Step 9:** Open the **application.properties** file and disable the security feature of the actuator by adding the following statement.

**application.properties**

1. management.security.enabled=false

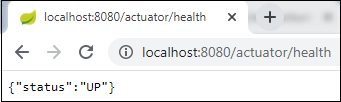
**Step 10:** Run the **SpringBootActuatorExampleApplication.java** file.

**Step 11:** Open the browser and invoke the URL http://localhost:8080/actuator. It returns the following page:

1. {"\_links":{"self":{"href":"http://localhost:8080/actuator","templated":false},"health":{"href":"http://localhost:8080/actuator/health","templated":false},"health-path":{"href":"http://localhost:8080/actuator/health/{\*path}","templated":true},"info":{"href":"http://localhost:8080/actuator/info","templated":false}}}

The application runs on port 8080 by default. Once the actuator has started, we can see the list of all the endpoints exposed over HTTP.

Let's invoke the **health** endpoint by invoking the URL http://localhost:8080/actuator/health. It denotes the status **UP**. It means the application is healthy and running without any interruption.



Similarly, we can invoke other endpoints that helps us to monitor and manage the Spring Boot application.

# Spring Boot Starter Test

The **spring-boot-starter-test** is the primary dependency for the test. It contains the majority of elements required for our tests.

There are several different types of tests that we can write to help test and automate the health of an application. Before starting any testing, we need to integrate the testing framework.

With Spring Boot, we need to add **starter** to our project, for testing we only need to add the **spring-boot-starter-test** dependency.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-test**</artifactId>**
4. **<version>**2.2.2.RELEASE**</version>**
5. **<scope>**test**</scope>**
6. **</dependency>**

It pulls all the dependencies related to test. After adding it, we can build up a simple unit test. We can either create the Spring Boot project through IDE or generate it using Spring Initializr.

10 Sec

Hello Java Program for Beginners

#### **Note: If you are adding test dependency manually, add it to the bottom of the pom.xml file.**

In the above dependency, one thing to be noticed that it includes the scope of test **<scope>test</scope>.** It means when the application is bundled and packaged for deployment, any dependency that is declared with the test scopes is ignored. The test scope dependencies are only available when running in the development and Maven test modes.

When we create a simple Spring Boot application, by default, it contains the test dependency in the pom.xml file and **ApplicationNameTest.java** file under in the folder **src/test/java.**

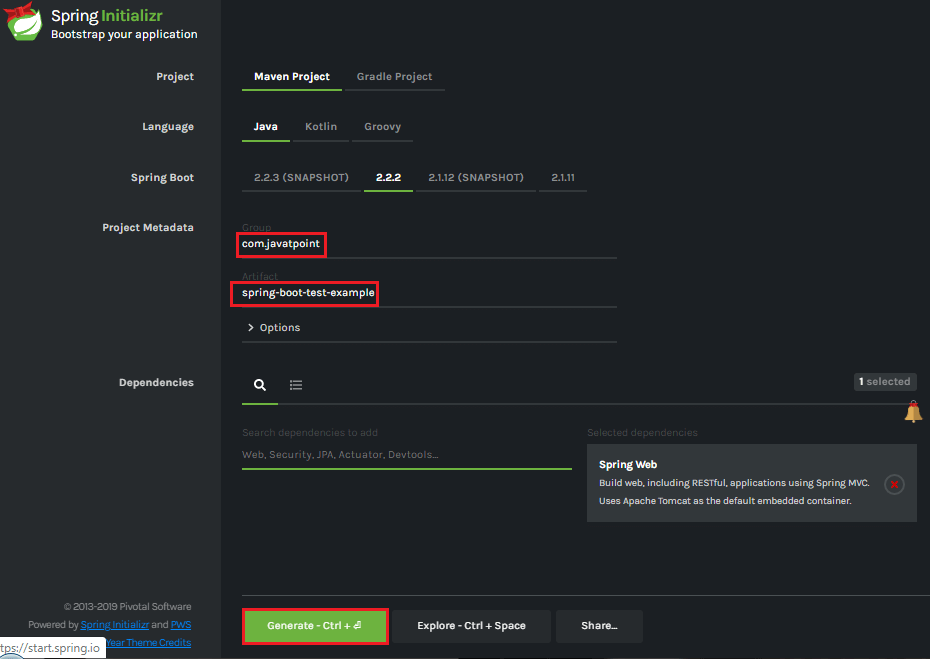
Let's create a simple maven project.

## Spring Boot Starter Test Example

**Step 1:** Open Spring Initializr [https://start.spring.io/.](https://start.spring.io/)

**Step 2:** Provide the **Group** name and **Artifact** Id. We have provided Group name **com.javatpoint** and Artifact **spring-boot-test-example.**

**Step 3:** Add the Spring Web dependency.



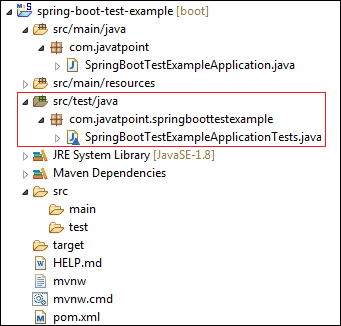
**Step 4:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications related to the project and downloads a **Jar** file to our local system.

**Step 5:** Extract the downloaded Jar file.

**Step 6:** Import the folder to STS. It takes some time to import.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-test-example -> Finish

After importing the project, we can see the following project directory in the Package Explorer section of the STS.



We can see in the above directory that it contains a test file named **SpringBootTestExampleApplicationTest.java** in the folder **src/test/java.**

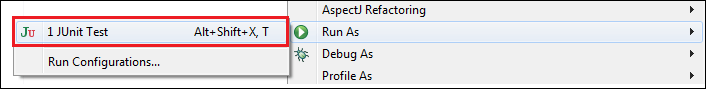
**SpringBootTestExampleApplicationTest.java**

1. **package** com.javatpoint.springboottestexample;
2. **import** org.junit.jupiter.api.Test;
3. **import** org.springframework.boot.test.context.SpringBootTest;
4. @SpringBootTest
5. **class** SpringBootTestExampleApplicationTests
6. {
7. @Test
8. **void** contextLoads()
9. {
10. }
11. }

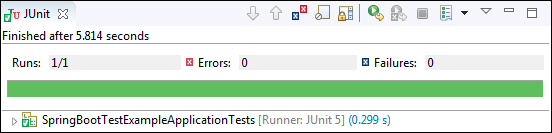
The above code implements **two** annotation by default: **@SpringBootTest,** and **@Test.**

* **@SpringBootTest:** It applies on a Test Class that runs Spring Boot based tests. It provides the following features over and above the regular Spring TestContext Framework:
  + It uses **SpringBootContextLoader** as the default ContextLoader if no specific @ContextConfiguration(loader=...) is defined.
  + It automatically searches for a **@SpringBootConfiguration** when nested @Configuartion is not used, and no explicit classes are specified.
  + It provides support for different **WebEnvironment** modes.
  + It registers a **TestRestTemplate** or WebTestClient bean for use in web tests that are using the webserver.
  + It allows application arguments to be defined using the **args attribute.**

**Step 7:** Open the **SpringBootTestExampleApplicationTest.java** file and run it as **Junit Test.**



When we run the above code, it displays the following:



# Spring Boot DevTools

## Spring Boot DevTools

Spring Boot 1.3 provides another module called Spring Boot DevTools. DevTools stands for **Developer Tool.** The aim of the module is to try and improve the development time while working with the Spring Boot application. Spring Boot DevTools pick up the changes and restart the application.

We can implement the DevTools in our project by adding the following dependency in the pom.xml file.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-devtools**</artifactId>**
4. **<scope>**runtime**<scope** **>**
5. **</dependency>**

## Spring Boot DevTools Features

Spring Boot DevTools provides the following features:

* **Property Defaults**
* **Automatic Restart**
* **LiveReload**
* **Remote Debug Tunneling**
* **Remote Update and Restart**

**Property Defaults:** Spring Boot provides templating technology **Thymeleaf** that contains the property **spring.thymeleaf.cache.** It disables the caching and allows us to update pages without the need of restarting the application. But setting up these properties during the development always creates some problems.

32.5M

562

Hello Java Program for Beginners

When we use the spring-boot-devtools module, we are not required to set properties. During the development caching for Thymeleaf, Freemarker, Groovy Templates are automatically disabled.

#### **Note: If we do not want to apply property defaults on an application, we can set configprop:spring.devtools.add-properties[] to false in the application.properties file.**

**Automatic Restart:** Auto-restart means reloading of Java classes and configure it at the server-side. After the server-side changes, it deployed dynamically, server restarts happen, and load the modified code. It is mostly used in microservice-based applications. Spring Boot uses **two** types of ClassLoaders:

* The classes that do not change (third-Jars) are loaded in the **base ClassLoader.**
* The classes that we are actively developing are loaded in the **restart ClassLoader.**

When the application restarts, the restart ClassLoader is thrown away, and a new one is populated. Therefore, the base ClassLoader is always available and populated.

We can disable the auto-restart of a server by using the property **spring.devtools.restart.enabled** to **false.**

### **Remember:**

* The DevTools always monitors the classpath resources.
* There is only a way to trigger a restart is to update the classpath.
* DevTools required a separate application classloader to work properly. By default, Maven fork the application process.
* Auto-restart works well with **LiveReload.**
* DevTools depends on the application context's shutdown hook to close it during the restart.

**LiveReload:** The Spring Boot DevTools module includes an embedded server called **LiveReload.** It allows the application to automictically trigger a browser refresh whenever we make changes in the resources. It is also known as **auto-refresh.**

**Note:** We can disable the LiveReload by setting the property **spring.devtools.livereload.enabled** to **false.**

It provides browser extensions for Chrome, Firefox, and Safari. By default, LiveReload is enabled. The LiveReload works on the following path:

* /META-INF/maven
* /META-INF/resources
* /resources
* /static
* /public
* /templates

We can also disable auto-reload in browser by excluding the above paths. For example:

1. spring.devtools.restart.exclude=public/\*\*, static/\*\*, templates/\*\*

We can see the other additional path by using the property **spring.devtools.restart.additional-paths.** For example:

1. spring.devtools.restart.additional-paths=/path-to-folder

If we want to exclude additional path and want to keep defaults then use the property **spring.devtools.restart.additional-exclude.** For example:

1. spring.devtools.restart.additional-exclude=styles/\*\*

### **Remember**

* We can run one LiveReload server at a time.
* Before starting the application, ensure that no other LiveReload server is running.
* If we start multiple applications from IDE, it supports only the first LiveReload.

**Remote Debug Tunneling:** Spring Boot can tunnel JDWP (Java Debug Wire Protocol) over HTTP directly to the application. It can even work application deployment to Internet Cloud providers that only expose port 80 and 443.

**Remote Update and Restart:** There is another trick that DevTools offers is: it supports remote application **updates** and **restarts.** It monitors local classpath for file changes and pushes them to a remote server, which is then restarted. We can also use this feature in combination with LiveReload.

## Using a Trigger File

Automatic restart sometimes can slow down development time due to frequent restarts. To remove this problem, we can use a **trigger file.** Spring Boot monitors trigger file and detects modifications in that file. It restarts the server and reloads all previous changes.

We can implement the trigger file in our application by adding the property **spring.devtools.restart.trigger-file.** The file can be internal or external. For example:

1. spring.devtools.restart.trigger-file=c:/workspace-sts-3.9.9.RELEASE/restart-trigger.txt

### **Spring Boot DevTools Example**

**Step 1:** Create a Maven project using Spring Initializr [https://start.spring.io/.](https://start.spring.io/)

**Step 2:** Provide the **Group** name and **Artifact** Id. We have provided Group name **com.javatpoint** and Artifact Id **spring-boot-devtools-example.**

**Step 3:** Add the following dependencies: **spring-boot-starter-web** and **spring-boot-devtools**.

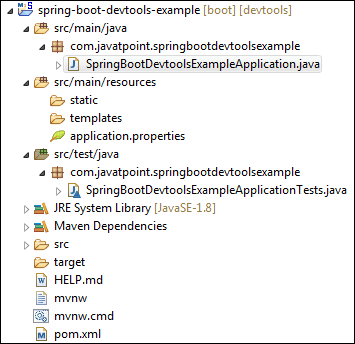
**Step 4:** Click on the **Generate** button. It downloads the **Jar** file of the project.

**Step 5:** Extract the Jar file.

**Step 6:** Import the folder to STS. It takes time to import.

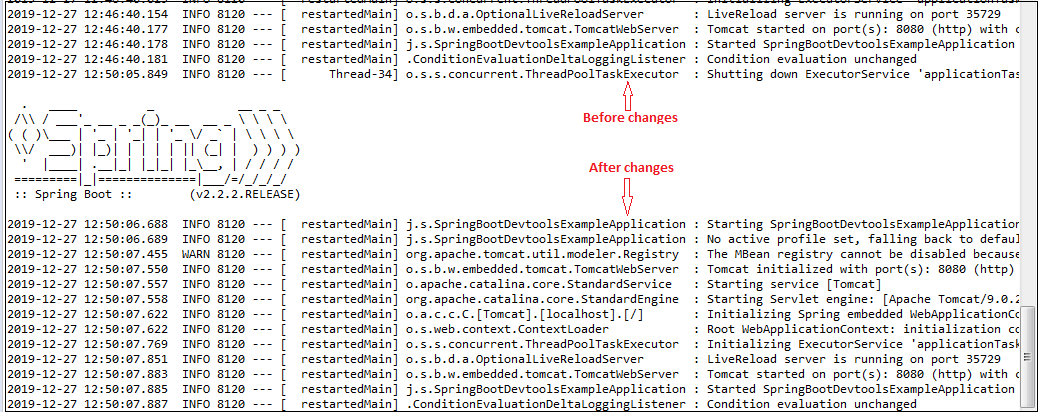
File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-devtools-example -> Finish

When the project imports successfully, we can see the following directory in the Package Explorer section of the STS.



**Step 7:** Open the **SpringBootDevtoolsExampleApplication.java** and run it as Java Application.

After that, make any changes (edit or remove some file or code) in the application, and save that changes. As soon as we save the changes, the server restarts and pick up the changes.



Spring Boot Multi-Module Project

Multi-Module Project

A Spring Boot project that contains nested maven projects is called the **multi-module project.** In the multi-module project, the parent project works as a container for base maven configurations.

In other words, a **multi-module project** is built from a parent pom that manages a group of submodules. Or A **multi-module project** is defined by a parent POM referencing one or more submodules.

The parent maven project must contain the packaging type **pom** that makes the project as an aggregator. The **pom.xml** file of the parent project consists the list of all **modules, common dependencies,** and **properties** that are inherited by the child projects. The parent pom is located in the project's root directory. The child modules are actual Spring Boot projects that inherit the maven properties from the parent project.

When we run the multi-module project, all the modules are deployed together in an embedded Tomcat Server. We can deploy an individual module, also.

Parent POM

The parent POM defines the **Group ID, Artifact ID, version**, and **packaging.** In the previous Maven projects, we have seen that the parent POM defines the packaging **jar.** But in the multi-module project, the parent **POM** defines the packaging pom. The packaging pom refers to other Maven projects.

Why we need multi-module project

Splitting the project into multiple modules is useful and easy to maintain. We can also easily edit or remove modules in the project without affecting the other modules. It is useful when we required to deploy modules individually.

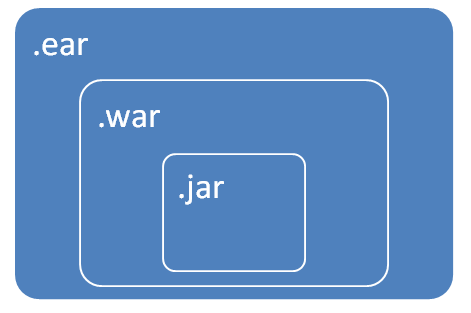
We only need to specify all the dependencies in the parent pom. All the other modules share the same pom, so we need not to specify the same dependency in each module separately. It makes the code easier to keep in order with a big project.

Child module-ear, war, and jar

The child module may be any project and can have any packaging. We are free to create any type of dependency between modules and bundles together.

For example, we are creating an **EAR** (Enterprise ARchive), **WAR** (Web ARchive), and **JAR** (Java ARchive) file. A JAR file is bundled into a war file that is bundled into an EAR file. The EAR file is the final package that is ready to deploy on the application server.

The EAR file contains one or many WAR files. Each WAR file contains the service project that has common code to all WAR files and packaging type in the JAR.



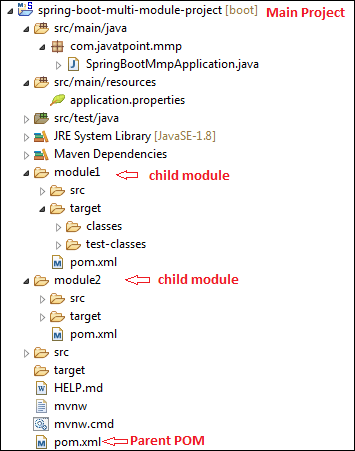
Maven child projects/ modules

* The child modules are independent maven projects that share properties from the parent project.
* All child projects can be built with a single command because it is inside a parent project.
* It is easier to define the relationship between the projects.

Multi-Module Project Directory Structure

Let's understand the multi-module project directory structure.

In the following image, we have created a project named **spring-boot-multi-module-project.** It contains the parent **pom** at the bottom of the directory. After that, we have created two **Maven Modules** named **module1** and **module2,** respectively. These two modules contain their own pom files.



Let's open the parent POM and see what it configures when we create Maven modules in the project.

**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" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**org.springframework.boot**</groupId>**
6. **<artifactId>**spring-boot-starter-parent**</artifactId>**
7. **<version>**2.2.2.BUILD-SNAPSHOT**</version>**
8. **<relativePath/>** <!-- lookup parent from repository -->
9. **</parent>**
10. **<groupId>**com.javatpoint**</groupId>**
11. **<artifactId>**spring-boot-example**</artifactId>**
12. **<version>**0.0.1-SNAPSHOT**</version>**
13. **<name>**spring-boot-multi-module-project**</name>**
14. **<description>**Demo project for Spring Boot**</description>**
15. **<properties>**
16. **<java.version>**1.8**</java.version>**
17. **</properties>**
18. **<packaging>**pom**</packaging>**
19. **<dependencies>**
20. **<dependency>**
21. **<groupId>**org.springframework.boot**</groupId>**
22. **<artifactId>**spring-boot-starter**</artifactId>**
23. **</dependency>**
24. **<dependency>**
25. **<groupId>**org.springframework.boot**</groupId>**
26. **<artifactId>**spring-boot-starter-parent**</artifactId>**
27. **<version>**2.2.1.RELEASE**</version>**
28. **<type>**pom**</type>**
29. **</dependency>**
30. **<dependency>**
31. **<groupId>**org.springframework.boot**</groupId>**
32. **<artifactId>**spring-boot-starter-web**</artifactId>**
33. **</dependency>**
34. **<dependency>**
35. **<groupId>**org.springframework**</groupId>**
36. **<artifactId>**spring-webmvc**</artifactId>**
37. **</dependency>**
38. **<dependency>**
39. **<groupId>**org.springframework.boot**</groupId>**
40. **<artifactId>**spring-boot-starter-test**</artifactId>**
41. **<scope>**test**</scope>**
42. **<exclusions>**
43. **<exclusion>**
44. **<groupId>**org.junit.vintage**</groupId>**
45. **<artifactId>**junit-vintage-engine**</artifactId>**
46. **</exclusion>**
47. **</exclusions>**
48. **</dependency>**
49. **</dependencies>**
50. **<build>**
51. **<plugins>**
52. **<plugin>**
53. **<groupId>**org.springframework.boot**</groupId>**
54. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
55. **</plugin>**
56. **</plugins>**
57. **</build>**
58. **<repositories>**
59. **<repository>**
60. **<id>**spring-milestones**</id>**
61. **<name>**Spring Milestones**</name>**
62. **<url>**https://repo.spring.io/milestone**</url>**
63. **</repository>**
64. **<repository>**
65. **<id>**spring-snapshots**</id>**
66. **<name>**Spring Snapshots**</name>**
67. **<url>**https://repo.spring.io/snapshot**</url>**
68. **<snapshots>**
69. **<enabled>**true**</enabled>**
70. **</snapshots>**
71. **</repository>**
72. **</repositories>**
73. **<pluginRepositories>**
74. **<pluginRepository>**
75. **<id>**spring-milestones**</id>**
76. **<name>**Spring Milestones**</name>**
77. **<url>**https://repo.spring.io/milestone**</url>**
78. **</pluginRepository>**
79. **<pluginRepository>**
80. **<id>**spring-snapshots**</id>**
81. **<name>**Spring Snapshots**</name>**
82. **<url>**https://repo.spring.io/snapshot**</url>**
83. **<snapshots>**
84. **<enabled>**true**</enabled>**
85. **</snapshots>**
86. **</pluginRepository>**
87. **</pluginRepositories>**
88. **<modules>**
89. **<module>**module1**</module>**
90. **<module>**module2**</module>**
91. **</modules>**
92. **</project>**

The above pom file is same as we have seen in the previous examples. But in this **pom** file, two things to be noticed: **packaging** and **modules.**

When we create multi-module project, we need to configure packaging pom in the parent pom file instead of **jar.**

1. **<packaging>**pom**</packaging>**

When we create Maven Modules in the project, Spring Boot automatically configures the modules in the parent pom inside the **module** tag, as shown below.

1. **<modules>**
2. **<module>**module1**</module>**
3. **<module>**module2**</module>**
4. **</modules>**

Now, we are going to see what inside the pom file of **module1.**

**pom.xml**

1. **<?xml** version="1.0"**?>**
2. **<project** xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**>**
4. **<modelVersion>**4.0.0**</modelVersion>**
5. **<parent>**
6. **<groupId>**com.javatpoint**</groupId>**
7. **<artifactId>**spring-boot-multi-module-project**</artifactId>**
8. **<version>**0.0.1-SNAPSHOT**</version>**
9. **</parent>**
10. **<groupId>**com.javatpoint**</groupId>**
11. **<artifactId>**module1**</artifactId>**
12. **<version>**0.0.1-SNAPSHOT**</version>**
13. **<name>**module1**</name>**
14. **<url>**http://maven.apache.org**</url>**
15. **<properties>**
16. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
17. **</properties>**
18. **<dependencies>**
19. **<dependency>**
20. **<groupId>**junit**</groupId>**
21. **<artifactId>**junit**</artifactId>**
22. **<version>**3.8.1**</version>**
23. **<scope>**test**</scope>**
24. **</dependency>**
25. **</dependencies>**
26. **</project>**

Here, a point to be noticed that the above pom file does not contain the common dependencies like **starter-web, web-mvc,** etc. It inherits all the common dependencies and the properties from the **parent pom.**

Spring Boot Multi-Module Project Example

Let' create an example of a multi-module application.

* **In the following example, we have created a maven project named spring-boot-multimodule. It is the main application. In the main application, we have created five modules are as follows:**
* **application**
* **model**
* **repository**
* **service-api**
* **service-impl**

**Application Module**

The application module is the main module of the project. It contains the application class in which the main method is defined that is necessary to run the Spring Boot Application. It also contains **application configuration properties, Controller, views,** and **resources.**

The Application Module includes Model Module, Service Implementation Module as dependency that contains Model Module, Repository Module, and Service API module.

**Model Module**

The Model Module contains **Entities** and **Visual Objects**to be used in the project.

**Repository Module**

The Repository module contains **repositories** to be used in the project. It depends on the Model Module.

**Service API Module**

The Service API module contains all project **services.** It also depends on Model Module.

**Service Implementation Module**

The Service Implementation module implements the service. It depends on Repository Module and Service API Module.

POM Aggregator (Parent POM)

The parent pom contains all the application modules. It also includes all the common dependencies and properties that are needed by more than one module. Dependencies are defined without version because the project has defined the Spring IO Platform as a parent.

Let's understand the structure of the multi-module application that we have created.

1. Spring-boot-multimodule
2. ├── pom.xml
3. │   └── REDME.adoc
4. ├── application
5. │   ├── pom.xml
6. │   └── src
7. │       └── main
8. │           ├── java
9. │           │   └── sample
10. │           │       └── multimodule
11. │           │           ├── SampleWebJspApplication.java
12. │           │           └── web
13. │           │               └── WelcomeController.java
14. │           └── resources
15. │               ├── application.properties
16. │               └── templates
17. │                   └── welcome
18. │                       └── show.html
19. ├── model
20. │   ├── pom.xml
21. │   └── src
22. │       └── main
23. │           └── java
24. │               └── sample
25. │                   └── multimodule
26. │                       └── domain
27. │                           └── entity
28. │                               └── Account.java
29. |
30. ├── repository
31. │   ├── pom.xml
32. │   └── src
33. │       └── main
34. │           └── java
35. │               └── sample
36. │                   └── multimodule
37. │                       └── repository
38. │                           └── AccountRepository.java
39. ├── service-api
40. │   ├── pom.xml
41. │   └── src
42. │       └── main
43. │           └── java
44. │               └── sample
45. │                   └── multimodule
46. │                       └── service
47. │                           └── api
48. │                               ├── AccountNotFoundException.java
49. │                               └── AccountService.java
50. └── service-impl
51. ├── pom.xml
52. └── src
53. └── main
54. └── java
55. └── sample
56. └── multimodule
57. └── service
58. └── impl
59. └── AccountServiceImpl.java

**Step 1:** Create a **Maven Project** with the name **spring-boot-multimodule.**

**Step 2:** Open the **pom.xml** (parent pom) file and change the packaging type **jar** to **pom.**

**pom.xml (parent pom)**

1. **<?xml** version="1.0" encoding="UTF-8" standalone="no"**?>**
2. **<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"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. <!-- Spring IO Platform is the parent of the generated application to
5. be able to use Spring Boot and all its default configuration --**>**
6. **<parent>**
7. **<groupId>**io.spring.platform**</groupId>**
8. **<artifactId>**platform-bom**</artifactId>**
9. **<version>**2.0.1.RELEASE**</version>**
10. **</parent>**
11. **<groupId>**sample.multimodule**</groupId>**
12. **<artifactId>**sample.multimodule**</artifactId>**
13. **<version>**0.0.1-SNAPSHOT**</version>**
14. **<packaging>**pom**</packaging>**
15. **<name>**Parent - Pom Aggregator**</name>**
16. **<properties>**
17. **<java.version>**1.8**</java.version>**
18. **</properties>**
19. **<dependencies>**
20. <!-- Spring Boot dependencies -->
21. **<dependency>**
22. **<groupId>**org.springframework.boot**</groupId>**
23. **<artifactId>**spring-boot-starter**</artifactId>**
24. **</dependency>**
25. **<dependency>**
26. **<groupId>**org.springframework.boot**</groupId>**
27. **<artifactId>**spring-boot-starter-data-jpa**</artifactId>**
28. **</dependency>**
29. **<dependency>**
30. **<groupId>**org.springframework.boot**</groupId>**
31. **<artifactId>**spring-boot-starter-test**</artifactId>**
32. **<scope>**test**</scope>**
33. **</dependency>**
34. **</dependencies>**
35. **</project>**

One thing to be noticed in the above pom file is that there is no maven module configured because we have not created yet. Now we will create Maven Modules one by one that we have specified above.

**Step 3:** Create a **Maven Module** with the name **application.**

**Step 4:** Open the **pom.xml** file of application module and ensure that the packaging type is **jar.**

**pom.xml**

1. **<?xml** version="1.0" encoding="UTF-8" standalone="no"**?>**
2. **<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"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**sample.multimodule**</groupId>**
6. **<artifactId>**sample.multimodule**</artifactId>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **</parent>**
9. **<artifactId>**sample.multimodule.application**</artifactId>**
10. **<packaging>**jar**</packaging>**
11. **<name>**Project Module - Application**</name>**
12. **<dependencies>**
13. <!-- Project modules -->
14. **<dependency>**
15. **<groupId>**sample.multimodule**</groupId>**
16. **<artifactId>**sample.multimodule.service.impl**</artifactId>**
17. **<version>**${project.version}**</version>**
18. **</dependency>**
20. <!-- Spring Boot dependencies -->
21. **<dependency>**
22. **<groupId>**org.apache.tomcat.embed**</groupId>**
23. **<artifactId>**tomcat-embed-jasper**</artifactId>**
24. **<scope>**provided**</scope>**
25. **</dependency>**
26. **<dependency>**
27. **<groupId>**org.springframework.boot**</groupId>**
28. **<artifactId>**spring-boot-starter-web**</artifactId>**
29. **</dependency>**
30. **<dependency>**
31. **<groupId>**org.springframework.boot**</groupId>**
32. **<artifactId>**spring-boot-starter-thymeleaf**</artifactId>**
33. **</dependency>**
35. **</dependencies>**
37. **<build>**
38. **<plugins>**
39. <!-- Spring Boot plugins -->
40. **<plugin>**
41. **<groupId>**org.springframework.boot**</groupId>**
42. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
43. **</plugin>**
44. **</plugins>**
45. **</build>**
47. **</project>**

**Step 5:** Create the **main** class. It is the class that is to be run.

**SampleWebJspApplication.java**

1. **package** sample.multimodule;
2. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
3. **import** org.springframework.boot.SpringApplication;
4. **import** org.springframework.boot.orm.jpa.EntityScan;
5. @SpringBootApplication
6. **public** **class** SampleWebJspApplication
7. {
8. **public** **static** **void** main(String[] args) **throws** Exception
9. {
10. SpringApplication.run(SampleWebJspApplication.**class**, args);
11. }
12. }

**Step 6:** Create a Controller class with the name **WelocameController** under the package **smaple.multimodule.web.**

**WelcomeController.java**

1. **package** sample.multimodule.web;
2. **import** java.util.Date;
3. **import** java.util.Map;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.beans.factory.annotation.Value;
6. **import** org.springframework.stereotype.Controller;
7. **import** org.springframework.web.bind.annotation.RequestMapping;
8. **import** sample.multimodule.domain.entity.Account;
9. **import** sample.multimodule.service.api.AccountService;
10. @Controller
11. **public** **class** WelcomeController
12. {
13. @Value("${application.message:Hello World}")
14. **private** String message = "Hello World";
15. @Autowired
16. **protected** AccountService accountService;
17. @RequestMapping("/")
18. **public** String welcome(Map<String, Object> model)
19. {
20. // Trying to obtain 23 account
21. Account account = accountService.findOne("23");
22. **if**(account == **null**){
23. // If there's some problem creating account, return show view with error status
24. model.put("message", "Error getting account!");
25. model.put("account", "");
26. **return** "welcome/show";
27. }
28. // Return show view with 23 account info
29. String accountInfo = "Your account number is ".concat(account.getNumber());
30. model.put("message", **this**.message);
31. model.put("account", accountInfo);
32. **return** "welcome/show";
33. }
34. @RequestMapping("foo")
35. **public** String foo(Map<String, Object> model) {
36. **throw** **new** RuntimeException("Foo");
37. }
38. }

**Step 7:** Create a **HTML** file with the name **show.html** under the folder **src/main/resource -> templates ->welcome.**

**show.html**

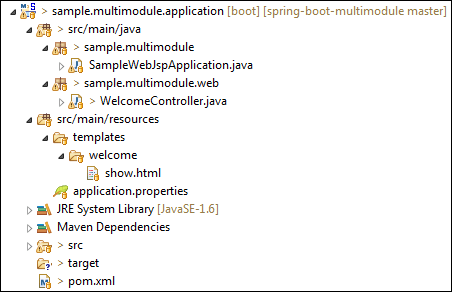
1. <!DOCTYPE HTML>
2. <html xmlns:th="http://www.thymeleaf.org">
3. <head>
4. <title>Spring Boot Multimodule</title>
5. <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6. </head>
7. <body>
8. <div>
9. <b>Message: </b>
10. <span th:text="${message}" />
11. </div>
12. <div>
13. <b>Your account: </b>
14. <span th:text="${account}" />
15. </div>
16. </body>
17. </html>

**Step 8:** Open the**application.properties** file, configure the **application message** and **thymeleaf** cache to **false.**

**application.properties**

1. # Application messages
2. application.message = Hello User!
3. dummy.type = type-inside-the-war
4. # Spring Thymeleaf config
5. spring.thymeleaf.cache = **false**

After creating all the above files, the application module directory looks like the following:



Let's create the second module that is **model.**

**Step 9:** Create a **Maven Module** with the name **model.**

**Step 10:** Open the **pom.xml** file of model module and ensure that the packaging type is **jar.**

**pom.xml**

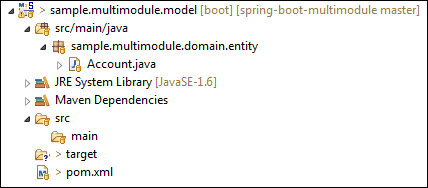
1. **<?xml** version="1.0" encoding="UTF-8" standalone="no"**?>**
2. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**sample.multimodule**</groupId>**
6. **<artifactId>**sample.multimodule**</artifactId>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **</parent>**
9. **<artifactId>**sample.multimodule.model**</artifactId>**
10. **<packaging>**jar**</packaging>**
11. **<name>**Project Module - Model**</name>**
12. **<description>**Module that contains all Entities and Visual Objects to be used in the project. It doesn't have any dependencies.
13. **</description>**
14. **</project>**

**Step 11:** Create a class with the name **Account** under the package **sample.multimodule.domain.entity.**

**Account.java**

1. **package** sample.multimodule.domain.entity;
2. **import** javax.persistence.Entity;
3. **import** javax.persistence.Id;
4. **import** javax.persistence.GeneratedValue;
5. **import** javax.persistence.GenerationType;
6. @Entity
7. **public** **class** Account
8. {
9. @Id
10. @GeneratedValue(strategy = GenerationType.AUTO)
11. **private** Long id;
13. **private** String number;
15. **private** String type;
17. **private** String creditCardNumber;
19. /\*\*
20. \* Create an empty account.
21. \*/
22. **public** Account() {
24. }
26. /\*\*
27. \* Create a new account.
28. \*
29. \* @param number
30. \*            the account number
31. \* @param id
32. \*            the account id
33. \*/
34. **public** Account(Long id, String number) {
35. **this**.number = number;
36. **this**.id = id;
37. }
39. **public** Long getId() {
40. **return** id;
41. }
43. **public** **void** setId(Long id) {
44. **this**.id = id;
45. }
47. **public** String getNumber() {
48. **return** number;
49. }
51. **public** **void** setNumber(String number) {
52. **this**.number = number;
53. }
55. **public** String getType() {
56. **return** type;
57. }
59. **public** **void** setType(String type) {
60. **this**.type = type;
61. }
63. **public** String getCreditCardNumber() {
64. **return** creditCardNumber;
65. }
67. **public** **void** setCreditCardNumber(String creditCardNumber) {
68. **this**.creditCardNumber = creditCardNumber;
69. }
71. }

After creating all the above files, the model module directory looks like the following:



Let's create the **third** module that is the **repository.**

**Step 12:** Create a **Maven Module** with the name **repository.**

**Step 13:** Open the **pom.xml** file of application module and ensure that the packaging type is **jar.**

**pom.xml**

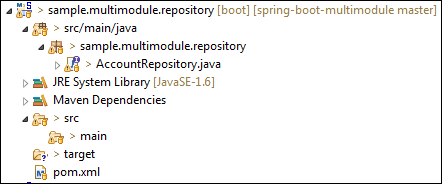
1. **<?xml** version="1.0" encoding="UTF-8" standalone="no"**?>**
2. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**sample.multimodule**</groupId>**
6. **<artifactId>**sample.multimodule**</artifactId>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **</parent>**
9. **<artifactId>**sample.multimodule.repository**</artifactId>**
10. **<packaging>**jar**</packaging>**
11. **<name>**Project Module - Repository**</name>**
12. **<description>**Module that contains all repositories to be used in the project. Depends of Model Module.**</description>**
14. **<dependencies>**
15. <!-- Project modules -->
16. **<dependency>**
17. **<groupId>**sample.multimodule**</groupId>**
18. **<artifactId>**sample.multimodule.model**</artifactId>**
19. **<version>**${project.version}**</version>**
20. **</dependency>**
22. <!-- Spring Boot dependencies -->
23. **<dependency>**
24. **<groupId>**org.hsqldb**</groupId>**
25. **<artifactId>**hsqldb**</artifactId>**
26. **<scope>**runtime**</scope>**
27. **</dependency>**
29. **</dependencies>**
31. **</project>**

**Step 14:** Create a class with the name **AccountRepository**under the package **sample.multimodule.repository.**

**AccountRepository.java**

1. **package** sample.multimodule.repository;
2. **import** org.springframework.data.domain.\*;
3. **import** org.springframework.data.repository.\*;
4. **import** org.springframework.stereotype.Repository;
5. **import** sample.multimodule.domain.entity.Account;
6. @Repository
7. **public** **interface** AccountRepository **extends** CrudRepository<Account, Long>
8. {
9. Account findByNumber(String number);
10. }

After creating all the above files, the repository module directory looks like the following:



Let's create the **fourth** module that is **service-api.**

**Step 15:** Create a **Maven Module** with the name **service-api.**

**Step 16:** Open the **pom.xml** file of application **service-api** and ensure that the packaging type is **jar.**

**pom.xml**

1. **<?xml** version="1.0" encoding="UTF-8" standalone="no"**?>**
2. **<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"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**sample.multimodule**</groupId>**
6. **<artifactId>**sample.multimodule**</artifactId>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **</parent>**
9. **<artifactId>**sample.multimodule.service.api**</artifactId>**
10. **<packaging>**jar**</packaging>**
11. **<name>**Project Module - Service API**</name>**
12. **<description>**Module that contains API of all project services. Depends of Model Module.**</description>**
14. **<dependencies>**
16. <!-- Project Modules -->
17. **<dependency>**
18. **<groupId>**sample.multimodule**</groupId>**
19. **<artifactId>**sample.multimodule.model**</artifactId>**
20. **<version>**${project.version}**</version>**
21. **</dependency>**
22. **</dependencies>**
23. **</project>**

**Step 17:** Create a package with the name **sample.multimodule.service.api.**

**Step 18:** Create a class with the name **AccountNotFoundException.** It handles the exception if the account is not found.

**AccountNotFoundException.java**

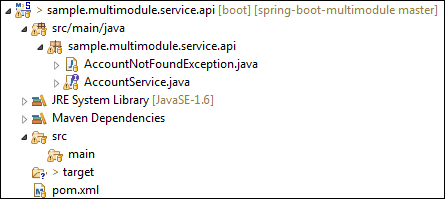
1. **package** sample.multimodule.service.api;
2. **public** **class** AccountNotFoundException **extends** RuntimeException
3. {
4. **private** **static** **final** **long** serialVersionUID = -3891534644498426670L;
5. **public** AccountNotFoundException(String accountId)
6. {
7. **super**("No such account with id: " + accountId);
8. }
9. }

**Step 19:** Create a class with the name **AccountService.** It provides the service related to account, such as **find** and **create** an account.

**AccountService.java**

1. **package** sample.multimodule.service.api;
2. **import** java.util.List;
3. **import** sample.multimodule.domain.entity.Account;
4. **public** **interface** AccountService
5. {
6. /\*\*
7. \* Finds the account with the provided account number.
8. \*
9. \* @param number The account number
10. \* @return The account
11. \* @throws AccountNotFoundException If no such account exists.
12. \*/
13. Account findOne(String number) **throws** AccountNotFoundException;
14. /\*\*
15. \* Creates a new account.
16. \* @param number
17. \* @return created account
18. \*/
19. Account createAccountByNumber(String number);
20. }

After creating all the above files, the service-api module directory looks like the following:



**Step 20:** Create a **Maven Module** with the name **service-impl.**

**Step 21:** Open the **pom.xml** file of application **service-impl** and ensure that the packaging type is jar.

**pom.xml**

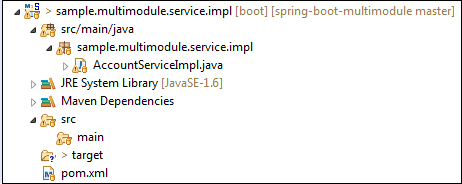
1. **<?xml** version="1.0" encoding="UTF-8" standalone="no"**?>**
2. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**sample.multimodule**</groupId>**
6. **<artifactId>**sample.multimodule**</artifactId>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **</parent>**
9. **<artifactId>**sample.multimodule.service.impl**</artifactId>**
10. **<packaging>**jar**</packaging>**
11. **<name>**Project Module - Service Implementation**</name>**
12. **<description>**Module that contains services implementation defined on Service API module. Depends of Repository Module and Service API Module.**</description>**
13. **<dependencies>**
14. <!-- Project Modules -->
15. **<dependency>**
16. **<groupId>**sample.multimodule**</groupId>**
17. **<artifactId>**sample.multimodule.repository**</artifactId>**
18. **<version>**${project.version}**</version>**
19. **</dependency>**
20. **<dependency>**
21. **<groupId>**sample.multimodule**</groupId>**
22. **<artifactId>**sample.multimodule.service.api**</artifactId>**
23. **<version>**${project.version}**</version>**
24. **</dependency>**
25. **</dependencies>**
26. **</project>**

**Step 22:** Create a class with the name **AccountServiceImpl** under the package **sample.multimodule.service.impl.**

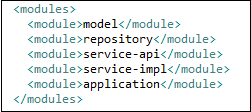
**AccountServiceImpl.java**

1. **package** sample.multimodule.service.impl;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Value;
5. **import** org.springframework.beans.factory.annotation.Autowired;
6. **import** org.springframework.stereotype.Service;
7. **import** sample.multimodule.domain.entity.Account;
8. **import** sample.multimodule.repository.AccountRepository;
9. **import** sample.multimodule.service.api.AccountService;
10. **import** sample.multimodule.service.api.AccountNotFoundException;
11. @Service
12. **public** **class** AccountServiceImpl **implements** AccountService
13. {
14. @Value("${dummy.type}")
15. **private** String dummyType;
16. @Autowired
17. **private** AccountRepository accountRepository;
18. /\*\*
19. \* {@inheritDoc}
20. \* <p/>
21. \* Dummy method for testing purposes.
22. \*
23. \* @param number The account number. Set 0000 to get an {@link AccountNotFoundException}
24. \*/
25. @Override
26. **public** Account findOne(String number) **throws** AccountNotFoundException {
27. **if**(number.equals("0000")) {
28. **throw** **new** AccountNotFoundException("0000");
29. }
30. Account account = accountRepository.findByNumber(number);
31. **if**(account == **null**){
32. account = createAccountByNumber(number);
33. }
34. **return** account;
35. }
36. @Override
37. **public** Account createAccountByNumber(String number) {
38. Account account = **new** Account();
39. account.setNumber(number);
40. **return** accountRepository.save(account);
41. }
42. **public** String getDummyType() {
43. **return** dummyType;
44. }
45. **public** **void** setDummyType(String dummyType) {
46. **this**.dummyType = dummyType;
47. }
48. }

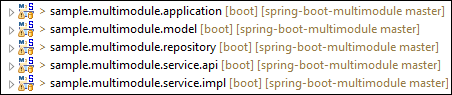
After creating all the above files, the **service-impl** module directory looks like the following:



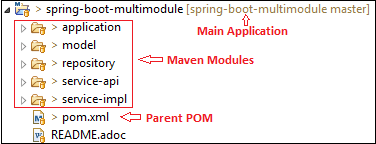
Now open the **parent pom** file, we see that all the Maven Modules that we have created is configured in the parent pom inside the tag. We need not to configure it manually.



Now ensure that all the five modules are created, as shown below:

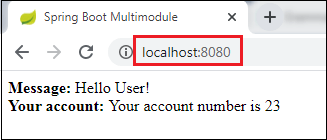


After creating all the modules, the main project directory looks like the following:



**Step 23:** Now run the **SampleWebJspApplication.java** file as Java Application.

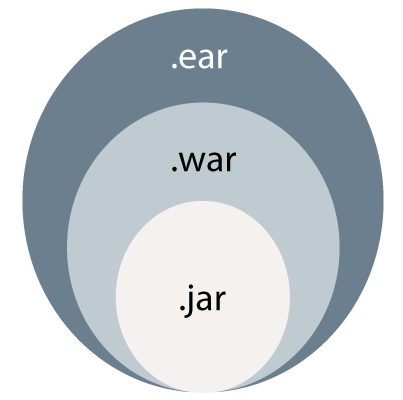
**Step 24:** Open the browser and invoke the URL http://localhost:8080. It returns the **Message** and account number that is **23.**



Spring Boot Packaging

In the J2EE application, modules are packed as **JAR, WAR,** and **EAR**. It is the compressed file formats that is used in the J2EE. J2EE defines three types of archives:

* WAR
* JAR
* EAR



WAR

**WAR** stands for **Web Archive.** WAR file represents the web application. Web module contains servlet classes, JSP files, HTML files, JavaScripts, etc. are packaged as a JAR file with .**war** extension. It contains a special directory called **WEB-INF**.

WAR is a module that loads into a web container of the Java Application Server. The Java Application Server has **two**containers: **Web Container** and **EJB Container**.

The **Web Container** hosts the web applications based on Servlet API and JSP. The web container requires the web module to be packaged as a WAR file. It is a WAR file special JAR file that contains a **web.xmlv** file in the **WEB-INF** folder.

OOPs Concepts in Java

An **EJB Container** hosts Enterprise Java beans based on EJB API. It requires EJB modules to be packaged as a JAR file. It contains an **ejb-jar.xml** file in the **META-INF** folder.

The advantage of the WAR file is that it can be deployed easily on the client machine in a Web server environment. To execute a WAR file, a Web server or Web container is required. For example, Tomcat, Weblogic, and Websphere.

JAR

**JAR** stands for **Java Archive.** An EJB (Enterprise Java Beans) module that contains bean files (class files), a manifest, and EJB deployment descriptor (XML file) are packaged as JAR files with the extension .**jar.** It is used by software developers to distribute Java classes and various metadata.

In other words, A file that encapsulates one or more Java classes, a manifest, and descriptor is called JAR file. It is the lowest level of the archive. It is used in J2EE for packaging EJB and client-side Java Applications. It makes the deployment easy.

EAR

**EAR** stands for **Enterprise Archive.** EAR file represents the enterprise application. The above two files are packaged as a JAR file with the .**ear** extension. It is deployed into the Application Server. It can contain multiple EJB modules (JAR) and Web modules (WAR). It is a special JAR that contains an **application.xml** file in the **META-INF** folder.

# Spring Boot Auto-configuration

Spring Boot auto-configuration automatically configures the Spring application based on the jar dependencies that we have added.

For example, if the H2 database Jar is present in the classpath and we have not configured any beans related to the database manually, the Spring Boot's auto-configuration feature automatically configures it in the project.

We can enable the auto-configuration feature by using the annotation **@EnableAutoConfiguration**. But this annotation does not use because it is wrapped inside the **@SpringBootApplication** annotation. The annotation @SpringBootApplication is the combination of three annotations: **@ComponentScan, @EnableAutoConfiguration,** and **@Configuration**. However, we use @SpringBootApplication annotation instead of using @EnableAutoConfiguration.

**@SpringBootApplication=@ComponentScan+@EnableAutoConfiguration+@Configuration**

26M

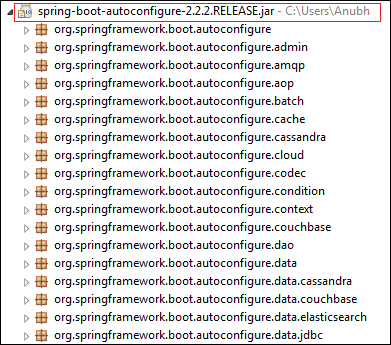
526

Java Try Catch

When we add the **spring-boot-starter-web** dependency in the project, Spring Boot auto-configuration looks for the Spring MVC is on the classpath. It auto-configures **dispatcherServlet**, a default **error page,** and **web jars**.

Similarly, when we add the **spring-boot-starter-data-jpa** dependency, we see that Spring Boot Auto-configuration, auto-configures a **datasource** and an **Entity Manager**.

All auto-configuration logic is implemented in **spring-boot-autoconfigure.jar**, as shown in the following figure.



## Need of auto-configuration

Spring-based application requires a lot of configuration. When we use Spring MVC, we need to configure **dispatcher servlet, view resolver, web jars** among other things. The following code shows typical configuration of a dispatcher servlet in a web application:

1. **<servlet>**
2. **<servlet-name>**dispatcher**</servlet-name>**
3. **<servlet-class>**
4. org.springframework.web.servlet.DispatcherServlet
5. **</servlet-class>**
6. **<init-param>**
7. **<param-name>**contextConfigLocation**</param-name>**
8. **<param-value>**/WEB-INF/todo-servlet.xml**</param-value>**
9. **</init-param>**
10. **<load-on-startup>**1**</load-on-startup>**
11. **</servlet>**
12. **<servlet-mapping>**
13. **<servlet-name>**dispatcher**</servlet-name>**
14. **<url-pattern>**/**</url-pattern>**
15. **</servlet-mapping>**

Similarly, when we use Hibernate/ JPA, we need to configure datasource, a transaction manager, an entity manager factory among a host of other things.

**Configuring datasource**

1. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
2. destroy-method="close"**>**
3. **<property** name="driverClass" value="${db.driver}" **/>**
4. **<property** name="jdbcUrl" value="${db.url}" **/>**
5. **<property** name="user" value="${db.username}" **/>**
6. **<property** name="password" value="${db.password}" **/>**
7. **</bean>**
8. **<jdbc:initialize-database** data-source="dataSource"**>**
9. **<jdbc:script** location="classpath:config/schema.sql" **/>**
10. **<jdbc:script** location="classpath:config/data.sql" **/>**
11. **</jdbc:initialize-database>**

**Configuring entity manager factory**

1. **<bean**
2. class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
3. id="entityManagerFactory"**>**
4. **<property** name="persistenceUnitName" value="hsql\_pu" **/>**
5. **<property** name="dataSource" ref="dataSource" **/>**
6. **</bean>**

**Configuring transaction manager**

1. **<bean** id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"**>**
2. **<property** name="entityManagerFactory" ref="entityManagerFactory" **/>**
3. **<property** name="dataSource" ref="dataSource" **/>**
4. **</bean>**
5. **<tx:annotation-driven** transaction-manager="transactionManager"**/>**

## Disable Auto-configuration Classes

We can also disable the specific auto-configuration classes, if we do not want to be applied. We use the **exclude** attribute of the annotation @EnableAutoConfiguration to disable the auto-configuration classes. For example:

1. import org.springframework.boot.autoconfigure.\*;
2. import org.springframework.boot.autoconfigure.jdbc.\*;
3. import org.springframework.context.annotation.\*;
4. @Configuration(proxyBeanMethods = false)
5. @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
6. public class MyConfiguration
7. {
8. }

We can use the attribute **excludeName** of the annotation @EnableAutoConfiguration and specify the **qualified** name of the class, if the class is not on the class path. We can exclude any number of auto-configuration classes by using the property **spring.autoconfigure.exclude**.

### **Spring Boot Auto-configuration Example**

In the following example, we will see how Spring Boot's auto-configuration features work.

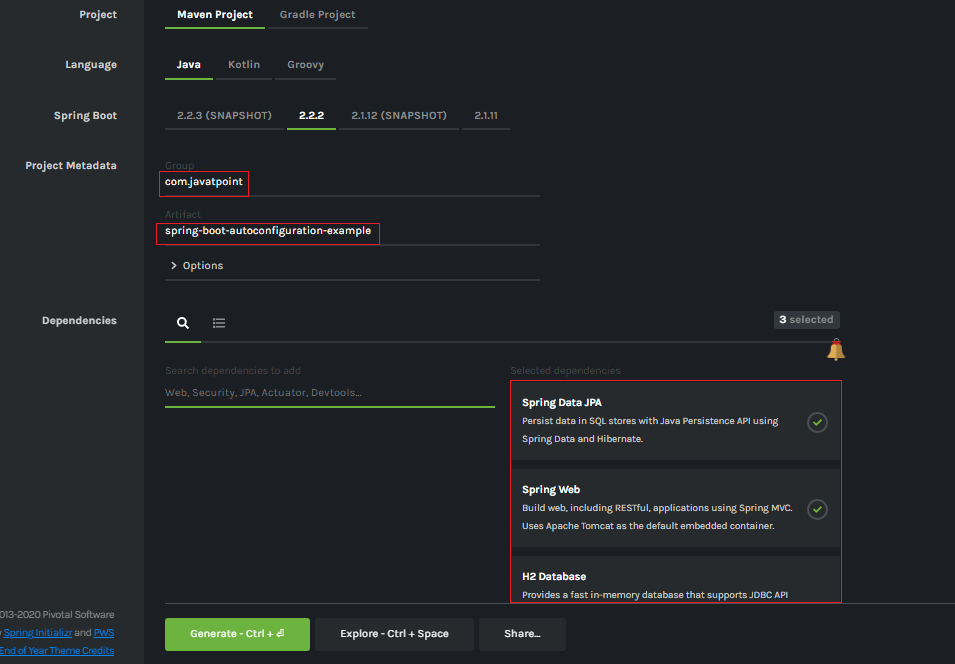
**Step 1:** Open spring Initializr <https://start.spring.io/>.

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint**.

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-autoconfiguration-example**.

**Step 4:** Add the dependencies: **Spring Web, Spring Data JPA,** an **H2 Database**.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps the project in **Jar** file and downloads it to the local system.



**Step 6: Extract** the Jar file and paste it into the STS workspace.

**Step 7: Import** the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-autoconfiguration-example -> Finish

It takes some time to import.

**Step 8:** Create a package with the name **com.javatpoint.controller**in the folder **src/main/java**.

**Step 9:** Create a Controller class with the name **ControllerDemo** in the package **com.javatpoint.controller**.

**ControllerDemo.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. @Controller
5. **public** **class** ControllerDemo
6. {
7. @RequestMapping("/")
8. **public** String home()
9. {
10. **return** "home.jsp";
11. }
12. }

**Step 10:** Create another package with the name **com.javatpoint.model** in the folder **src/main/java**.

**Step 11:** Create a class with the name **User** in the package **com.javatpoint.model**.

**User.java**

1. **package** com.javatpoint.model;
2. **import** javax.persistence.Entity;
3. **import** javax.persistence.Id;
4. **import** javax.persistence.Table;
5. @Entity
6. @Table(name="userdata")
7. **public** **class** User
8. {
9. @Id
10. **private** **int** id;
11. **private** String username;
12. **public** **int** getId()
13. {
14. **return** id;
15. }
16. **public** **void** setId(**int** id)
17. {
18. **this**.id = id;
19. }
20. **public** String getUname()
21. {
22. **return** username;
23. }
24. **public** **void** setUname(String username)
25. {
26. **this**.username = username;
27. }
28. @Override
29. **public** String toString()
30. {
31. **return** "User [id=" + id + ", uname=" + username + "]";
32. }
33. }

Now we need to Configure the H2 database.

**Step 12:** Open the **application.properties**file and configure the following things: **port, enable the H2 console, datasource,** and **URL.**

**application.properties**

1. server.port=8085
2. spring.h2.console.enabled=true
3. spring.datasource.plateform=h2
4. spring.datasource.url=jdbc:h2:mem:javatpoint

**Step 13:** Create a **SQL** file in the folder **src/main/resources.**

Right-click on the folder src/main/resources -> New -> File -> Provide the File name -> Finish

We have provided the file name **data.sql** and insert the following data into it.

**data.sql**

1. insert into userdata values(101,'Tom');
2. insert into userdata values(102,'Andrew');
3. insert into userdata values(103,'Tony');
4. insert into userdata values(104,'Bob');
5. insert into userdata values(105,'Sam');

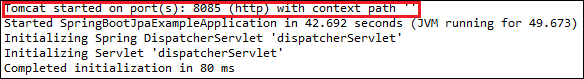
**Step 14:** Create a folder with the name **webapp** in the **src** folder.

**Step 15:** Create a JSP file with the name that we have returned in the **ControllerDemo**. In the ControllerDemo.java, we have returned **home.jsp**.

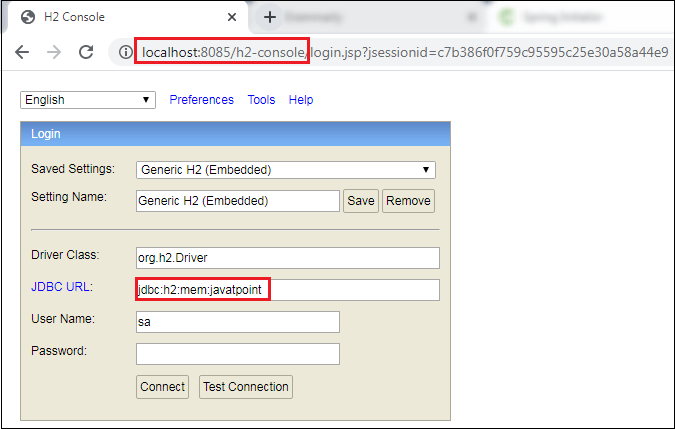
**home.jsp**

1. **<**%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2. pageEncoding="ISO-8859-1"%**>**
3. <!DOCTYPE html**>**
4. **<html>**
5. **<head>**
6. **<meta** charset="ISO-8859-1"**>**
7. **<title>**Insert title here**</title>**
8. **</head>**
9. **<body>**
10. **<form** action="addUser"**>**
11. ID :**<br** **/>**
12. **<input** type="text" name="t1"**><br** **/>**
13. User name :**<br** **/>**
14. **<input** type="text" name="t2"**><br** **/>**
15. **<input** type="submit" value="Add"**>**
16. **</form>**
17. **</body>**
18. **</html>**

**Step 16:** Run the **SpringBootAutoconfigurationExampleApplication.java**file. We can see in the console that our application is successfully running on port **8085**.

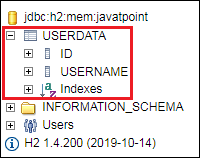


**Step 17:** Open the browser and invoke the URL http://localhost:8085/h2-console/. It shows the **Driver Class**, **JDBC URL** that we have configured in the **application.properties** file, and the default User Name **sa**.



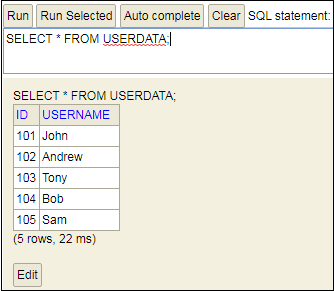
We can also test the connection by clicking on the **Test Connection**button. If the connection is successful, it shows a message **Test Successful.**

**Step 18:** Click on the **Connect** button. It shows the structure of the table **userdata**that we have defined in the User.java file.

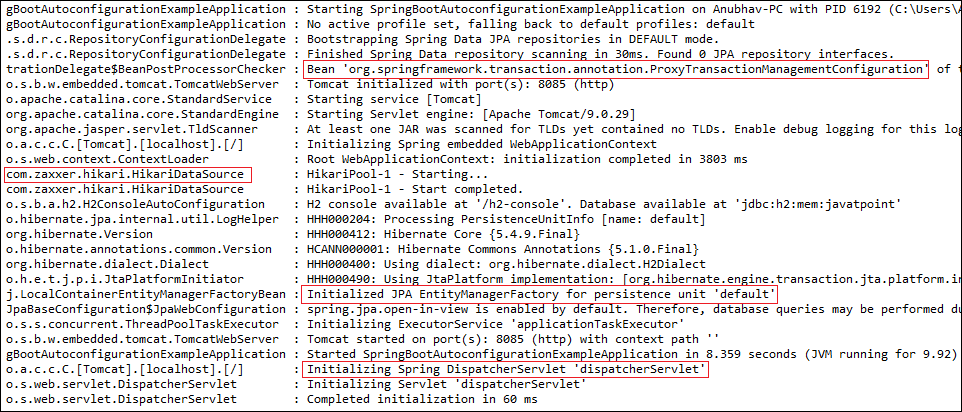


**Step 19:** Execute the following query to see the data that we have inserted in the **data.sql** file.

1. SELECT \* FROM USERDATA;



Let's have a close look at the console. We see that **TransactionManagement, DispatcherServlet, EntityManagerFactory,**and **DataSource** automatically configures, as shown in the following figure.



## Debugging Auto-configuration

We can find more information about auto-configuration by using the following two ways:

* Turning on debug logging
* Using Spring Boot Actuator

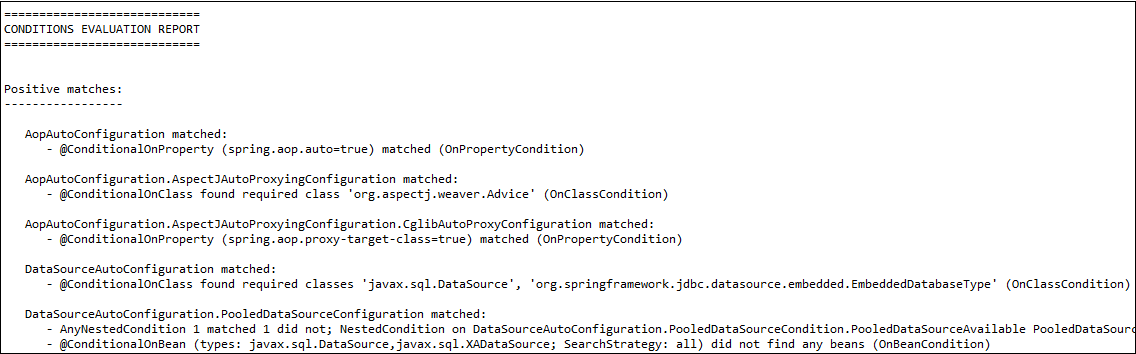
**Turning on debug logging**

We can debug logging by adding a property in the **application.properties** file. Let's implement the debug logging in the above example. Open the **application.properties** file and add the following property:

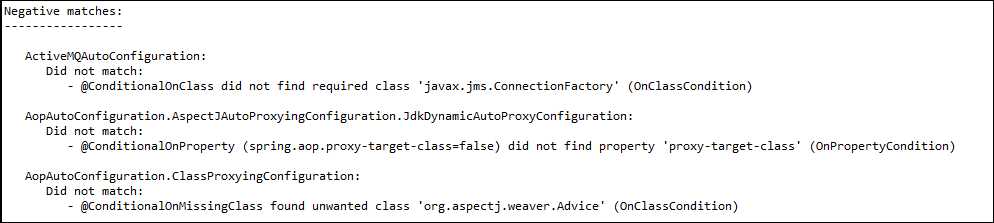
1. logging.level.org.springframework: DEBUG

Now restart the application. We see that an auto-configuration report printed in the log. The report includes all the classes that are auto-configured. It is divided into two parts: **positive matches** an**negative matches,**as shown in the following figure.

**Positive matches**



**Negative matches**



**Spring Boot Actuator**

We can also debug auto-configuration by using **Actuator** in the project. We will also add in **HAL Browser** to make things easy. It shows the details of all the beans that are auto-configured, and those are not.

Let's create an example of Spring Boot Actuator.

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided **actuator-autoconfiguration-example.**

**Step 4:** Add the dependencies: **Spring Web** and **Spring Boot Actuator.**

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **</dependency>**
5. **<dependency>**
6. **<groupId>**org.springframework.boot**</groupId>**
7. **<artifactId>**spring-boot-starter-actuator**</artifactId>**
8. **</dependency>**

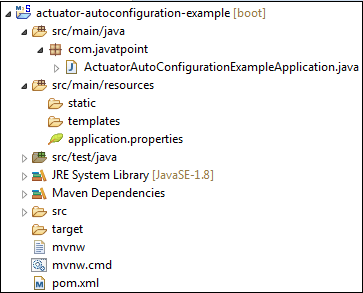
**Step 5:** Click on the **Generate** button. It binds all the specification related to project into a **jar** file and downloads it in our local system.

**Step 6: Extract** the downloaded jar file.

**Step 7: Import** the project folder by using the following steps:

File -> Import -> Existing Maven Project -> Next -> Browse -> Select the Project folder -> Finish

After importing the project, we can see the following directory structure in the **Package Explorer** section of the IDE.



**Step 8:** Create a Controller class in the package **com.javatpoint.** We have created a controller class with the name **DemoRestController.**

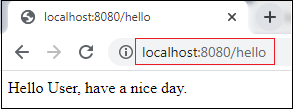
In the Controller, we have defined a method named **hello()** that returns a string.

**DemoRestController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. @RestController
5. **public** **class** DemoRestController
6. {
7. @GetMapping("/hello")
8. **public** String hello()
9. {
10. **return** "Hello User, have a nice day.";
11. }
12. }

**Step 9:** Run the**ActuatorAutoConfigurationExampleApplication.java** file.

**Step 10:** Open the browser and invoke the URL http://localhost:8080/hello. It returns a string that we have specified in the controller.



Now invoke the actuator URL http://localhost:8080/actuator. It launches the actuator that shows the three URLs: **self**, **health**, and **info,**as shown below.

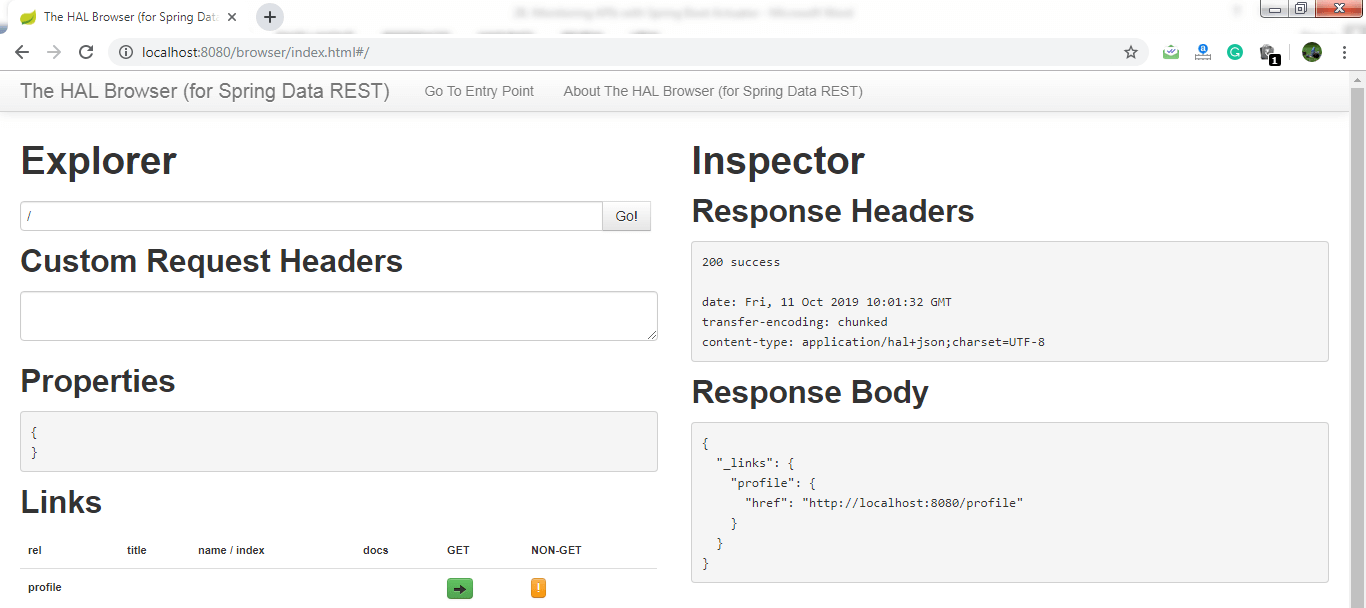
1. {"\_links":{"self":{"href":"http://localhost:8080/actuator","templated":false},"health":{"href":"http://localhost:8080/actuator/health","templated":false},"health-path":{"href":"http://localhost:8080/actuator/health/{\*path}","templated":true},"info":{"href":"http://localhost:8080/actuator/info","templated":false}}}

**Step 11:** Open the**pom.xml**file and add the **HAL Browser** dependency.

1. **<dependency>**
2. **<groupId>**org.springframework.data**</groupId>**
3. **<artifactId>**spring-data-rest-hal-browser**</artifactId>**
4. **</dependency>**

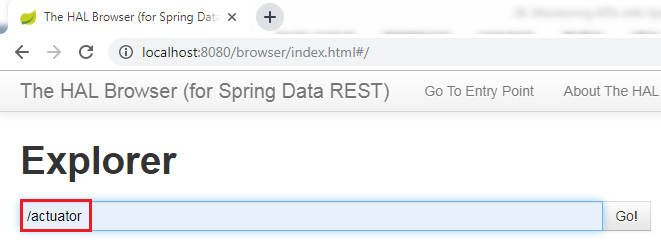
**Step 12:** Again, run the **ActuatorAutoConfigurationExampleApplication.java** file.

To access the HAL browser, type http://localhost:8080 in the browser and hit the enter key.



Now we can access the actuator through the HAL browser.

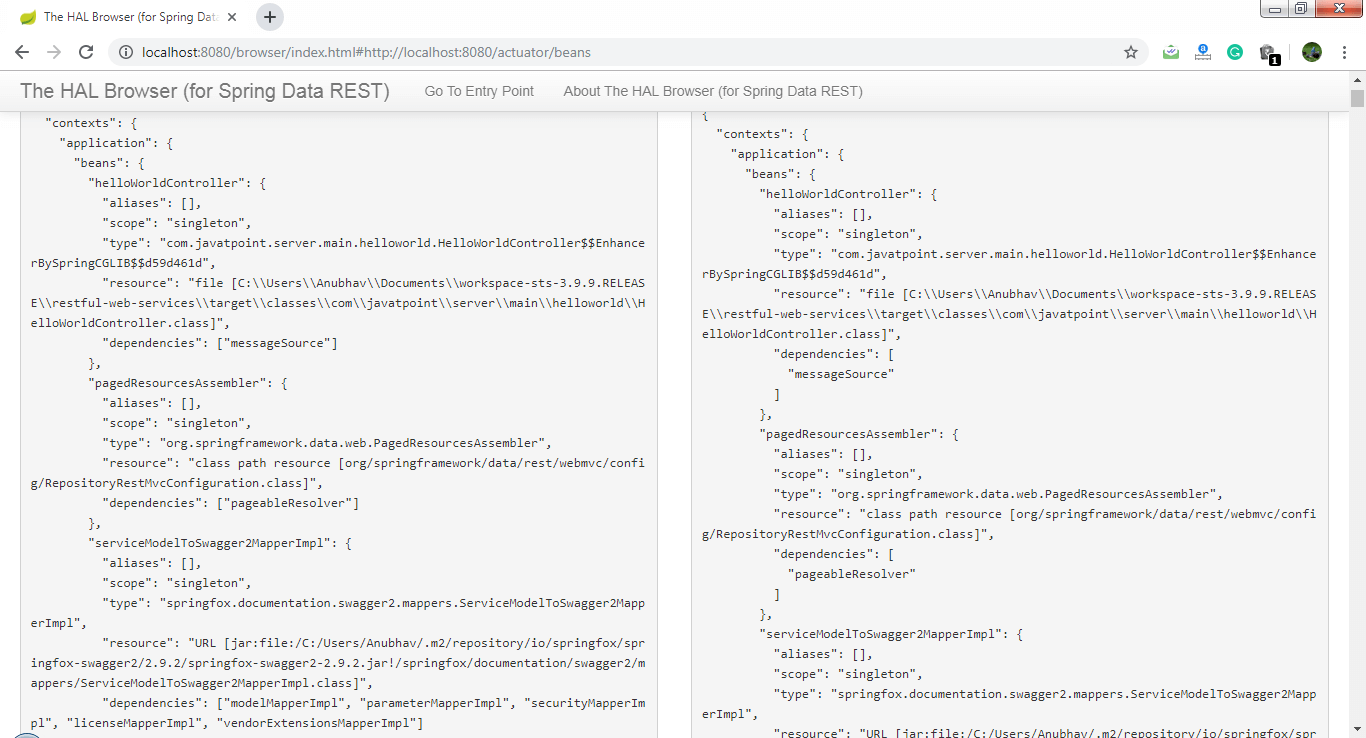
Type **/actuator** in the Explorer's text box and click on the **Go** button.



It shows all the things related to the actuator. The most important thing in the actuator is **beans**.



When we click on the bean's arrow, it shows all the beans configured in the Spring Boot project.



The above image shows the details of all the **beans** that are auto-configured and that are not.

Spring Boot Hello World Example

In the section, we will create a **Maven** project for Hello Word Example. We need the following tools and technologies to develop the same.

* Spring Boot 2.2.2.RELEASE
* JavaSE 1.8
* Maven 3.3.9
* STS IDE

**Step 1:** Open Spring Initializr <https://start.spring.io/>.

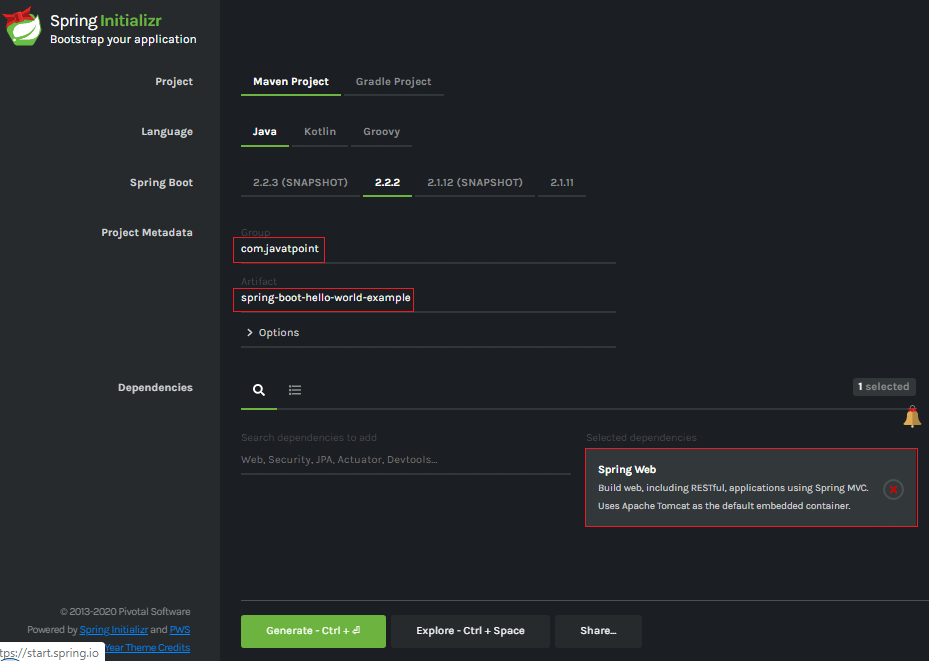
**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided the **spring-boot-hello-world-example.**

SQL CREATE TABLE

**Step 4:** Add the dependency **Spring Web.**

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications into a jar file and downloads it to our local system.

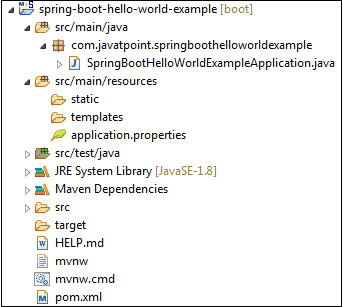


**Step 6: Extract** the RAR file.

**Step 7: Import** the project folder by using the following steps:

File -> Import -> Existing Maven Project -> Next -> Browse -> Select the Project Folder -> Finish

When the project imports successfully, it shows the following project directory in the Package Explorer section of the IDE.



**Step 8:** Create a package with the name **com.javatpoint.controller** inside the folder **src/main/java.**

**Step 9:** Create a Controller class with the name **HelloWorldController.**

**Step 10:** Create a method named **hello()**that returns a String.

**HelloWorldController.java**

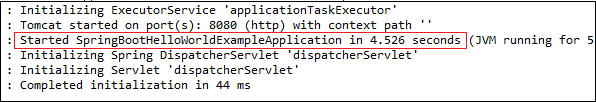
1. **package** com.javatpoint.controller;
2. **import** org.springframework.web.bind.annotation.RequestMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. @RestController
5. **public** **class** HelloWorldController
6. {
7. @RequestMapping("/")
8. **public** String hello()
9. {
10. **return** "Hello javaTpoint";
11. }
12. }

**Step 11:** Run the **SpringBootHelloWorldExampleApplication.java** file.

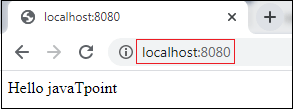
**SpringBootHelloWorldExampleApplication.java**

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

When the application runs successfully, it shows a massage in the console, as shown in the following figure.



**Step 12:** Open the browser and invoke the URL **https://localhost:8080**. It returns a String that we have specified in the Controller.



# Project Deployment Using Tomcat

In this section, we will learn how to deploy the Spring Boot application on Tomcat Server.

It includes three steps:

* **Setting up a Spring Boot Application**
* **Create a Spring Boot WAR**
* **Deploy the WAR to Tomcat**

### **Example**

Let's create a Maven example to be deployed on Tomcat

**Setting up a Spring Boot Application**

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

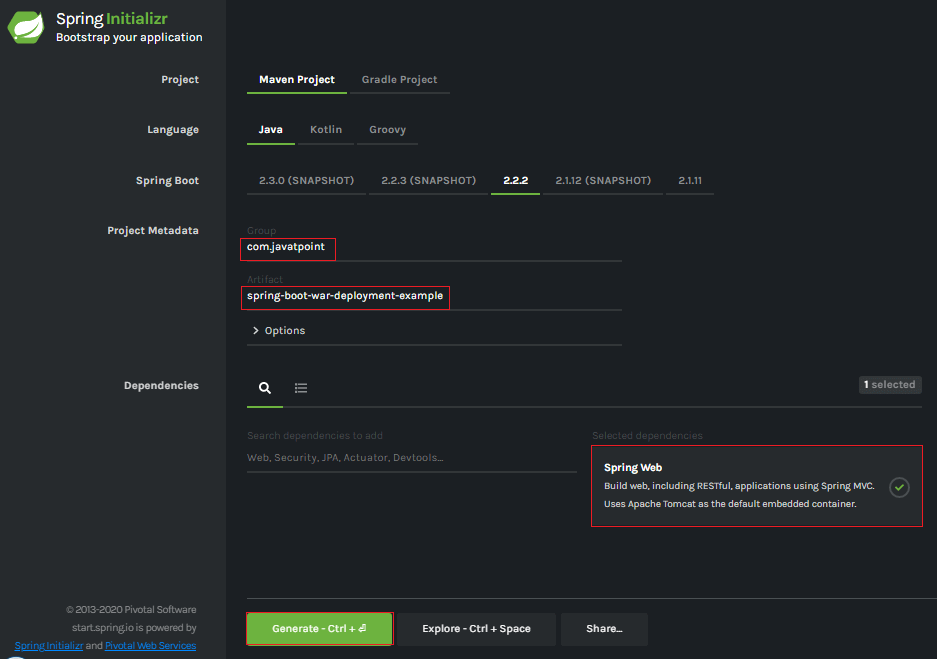
**Step 2:** Provide the **Group** name. We have provided the **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided the **spring-boot-war-deployment-example.**

**Step 4:** Add the **Spring Web** dependency.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **</dependency>**

**Step 5:** Click on the **Generate** button. It wraps all the specification related to the project and download the **jar** file in our local system.

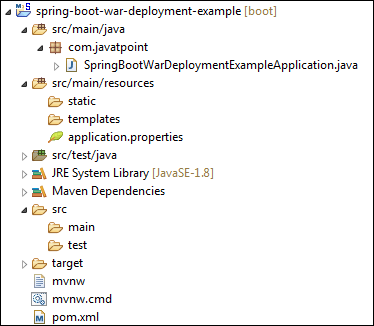


**Step 6: Extract** the jar file.

**Step 7: Import** the project folder by using the following steps:

File -> Import -> Existing Maven Project -> Next -> Browse -> Select the Project Folder -> Finish

After importing the project, we can see the following directory structure in the **Package Explorer** section of the IDE.



**Step 8:** Create a Controller class in the package **com.javatpoint**. We have created a class with the name **DemoRestController.**

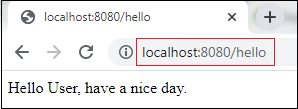
Inside the controller class, we have defined a method **hello()** that returns a String.

**DemoRestController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. @RestController
5. **public** **class** DemoRestController
6. {
7. @GetMapping("/hello")
8. **public** String hello()
9. {
10. **return** "Hello User, have a nice day.";
11. }
12. }

**Step 9:** Run the **SpringBootWarDeploymentExampleApplication.java** file as Java Application.

**Step 10:** Open the browser and invoke the URL http://localhost:8080/hello.



#### **Note: Before moving to the next step, ensure that the application is running correctly.**

## Create a Spring Boot WAR

It makes use of Spring Framework's Servlet 3.0 support and allows us to configure application when the servlet container launches it. There are **three** steps to create a WAR for deployment:

* Extends the **SpringBootServletInitializer**class in the main class.
* Marked the embedded servlet container as **provided**.
* Update packaging **JAR** to

Let's implement the above three steps in an application.

**Step 11:** Open the **SpringBootWarDeploymentExampleApplication.java**file and initialize the Servlet Context required by the Tomcat. To achieve the same extends the **SpringBootServletInitializer**interface.

1. **public** **class** SpringBootWarDeploymentExampleApplication **extends** SpringBootServletInitializer
2. {
3. }

**Step 12:** Override the **Configure** method.

1. @Override
2. **protected** SpringApplicationBuilder configure(SpringApplicationBuilder application)
3. {
4. **return** application.sources(SpringBootWarDeploymentExampleApplication.**class**);
5. }

**SpringBootWarDeploymentExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.boot.builder.SpringApplicationBuilder;
5. **import** org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
6. @SpringBootApplication
7. **public** **class** SpringBootWarDeploymentExampleApplication **extends** SpringBootServletInitializer
8. {
9. @Override
10. **protected** SpringApplicationBuilder configure(SpringApplicationBuilder application)
11. {
12. **return** application.sources(SpringBootWarDeploymentExampleApplication.**class**);
13. }
14. **public** **static** **void** main(String[] args)
15. {
16. SpringApplication.run(SpringBootWarDeploymentExampleApplication.**class**, args);
17. }
18. }

**Step 13:** Open the **pom.xml** file and marked the servlet container (Tomcat) as **provided**.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-tomcat**</artifactId>**
4. **<scope>**provided**</scope>**
5. **</dependency>**

**Step 14:** We need to deploy **WAR** file so change the package type to WAR in pom.xml file.

1. **<packaging>**war**</packaging>**

**Step 15:** Modify the final WAR file name by using the **<finalName>** tag to avoid including the version numbers. We have created a WAR file with the name **web-services.**

1. **<finalName>**web-services**</finalName>**

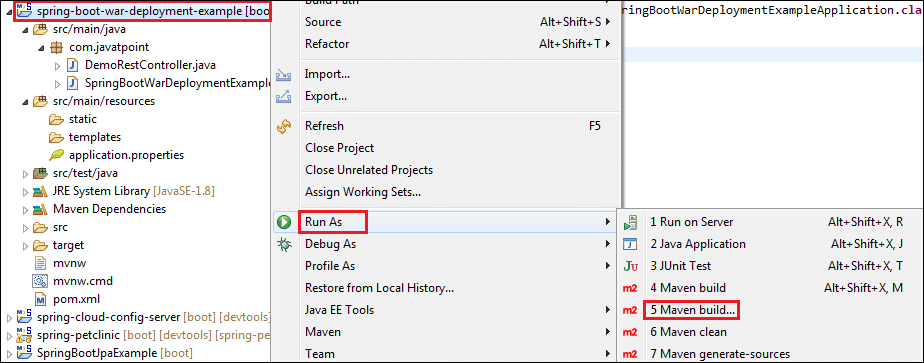
**pom.xml**

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. **<groupId>**com.javatpoint**</groupId>**
6. **<artifactId>**spring-boot-war-deployment-example**</artifactId>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **<packaging>**war**</packaging>**
9. **<name>**spring-boot-war-deployment-example**</name>**
10. **<description>**Demo project for Spring Boot**</description>**
11. **<parent>**
12. **<groupId>**org.springframework.boot**</groupId>**
13. **<artifactId>**spring-boot-starter-parent**</artifactId>**
14. **<version>**2.2.2.RELEASE**</version>**
15. **<relativePath/>** <!-- lookup parent from repository -->
16. **</parent>**
17. **<properties>**
18. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
19. **<project.reporting.outputEncoding>**UTF-8**</project.reporting.outputEncoding>**
20. **<java.version>**1.8**</java.version>**
21. **</properties>**
22. **<dependencies>**
23. **<dependency>**
24. **<groupId>**org.springframework.boot**</groupId>**
25. **<artifactId>**spring-boot-starter-web**</artifactId>**
26. **</dependency>**
27. **<dependency>**
28. **<groupId>**org.springframework.boot**</groupId>**
29. **<artifactId>**spring-boot-starter-tomcat**</artifactId>**
30. **<scope>**provided**</scope>**
31. **</dependency>**
32. **<dependency>**
33. **<groupId>**org.springframework.boot**</groupId>**
34. **<artifactId>**spring-boot-starter-test**</artifactId>**
35. **<scope>**test**</scope>**
36. **</dependency>**
37. **</dependencies>**
38. **<build>**
39. **<finalName>**web-services**</finalName>**
40. **<plugins>**
41. **<plugin>**
42. **<groupId>**org.springframework.boot**</groupId>**
43. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
44. **</plugin>**
45. **</plugins>**
46. **</build>**
47. **</project>**

To build our Tomcat-deployable WAR application, we execute the maven clean package. After that our WAR file is generated at **/target/abc.war** (where **abc** is assumed Artifact Id). We should consider that this new setup makes our Spring Boot Application a **non-standalone** application.

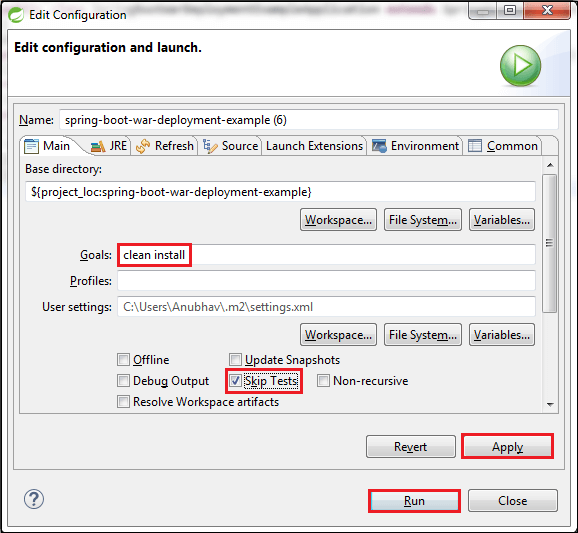
**Step 16:** Create a **WAR** file by using the following steps:

Right-click on the project -> Run As -> 5 Maven Build

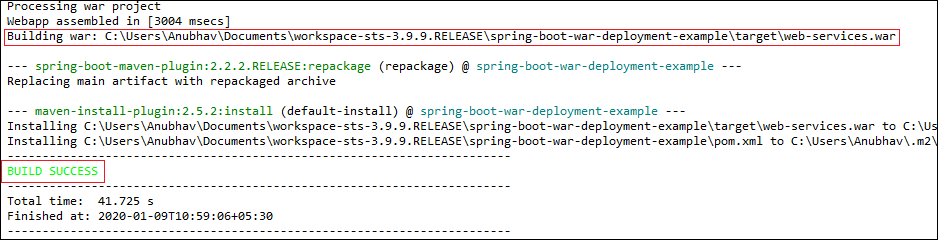


An **Edit Configuration** dialog box appears on the screen.

**Step 17:** Write **clean install** in the **Goals** label and check the **Skip Tests**. Click on the **Apply** and **Run**button, respectively.

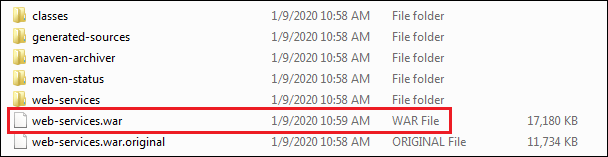


When the WAR file successfully created, it shows the **WAR file path** and a message **BUILD SUCCESS** in the console, as shown in the following figure.



**Step 18:** Copy the **path** and access the **target** folder of the application. We found the WAR file inside the target folder with the same name as we have specified in the pom.xml file. In our case, path is:

1. C:\Users\Anubhav\Documents\workspace-sts-3.9.9.RELEASE\spring-boot-war-deployment-example\target



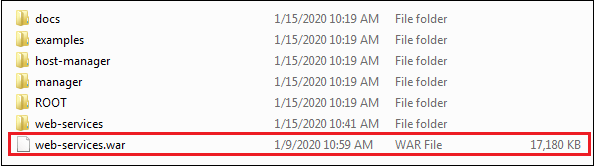
## Deploy the WAR file to Tomcat

To deploy the WAR file, follow the steps below:

**Step 19:** Download and install the **Apache Tomcat Server**, if not installed.

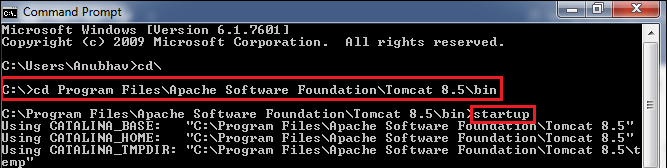
**Step 20:** Copy the WAR file **(web-services.war)** and paste it inside the **webapps** folder of the Tomcat. In our case, the location of the webapps folder is:

1. C:\Program Files\Apache Software Foundation\Tomcat 8.5\webapps

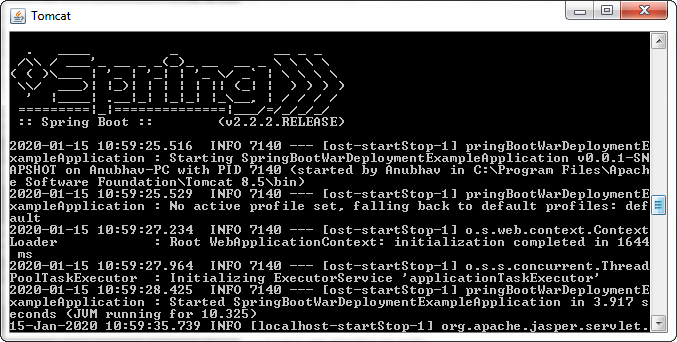


**Step 21:** Now open the Command Prompt and type the following commands:

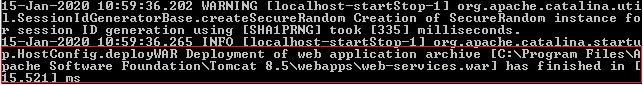
1. C:\Cd Program Files\Apache Software Foundation\Tomcat  8.5\bin
2. C:\Cd Program Files\Apache Software Foundation\Tomcat  8.5\bin**>**startup



The **startup** command starts the Tomcat server and deploys the WAR file, as shown below.



The following image shows that WAR is deployed successfully.



**Step 23:** Open the browser and invoke the URL http://localhost:8080/web-services/hello. It returns the message **Hello User, have a nice day**.

Spring Boot AOP

The application is generally developed with multiple layers. A typical Java application has the following layers:

* **Web Layer:** It exposes the **services** using the REST or web application.
* **Business Layer:** It implements the **business logic** of an application.
* **Data Layer:** It implements the **persistence logic** of the application.

The responsibility of each layer is different, but there are a few common aspects that apply to all layers are **Logging, Security, validation, caching,** etc. These common aspects are called **cross-cutting concerns.**

If we implement these concerns in each layer separately, the code becomes more difficult to maintain. To overcome this problem, **Aspect-Oriented Programming** (AOP) provides a solution to implement cross-cutting concerns.

* Implement the cross-cutting concern as an aspect.
* Define pointcuts to indicate where the aspect has to be applied.

It ensures that the cross-cutting concerns are defined in one cohesive code component.

31M

673

C++ vs Java

AOP

AOP **(Aspect-Oriented Programming)** is a programming pattern that increases modularity by allowing the separation of the **cross-cutting concern**. These cross-cutting concerns are different from the main business logic. We can add additional behavior to existing code without modification of the code itself.

Spring's AOP framework helps us to implement these cross-cutting concerns.

Using AOP, we define common functionality in one place. We are free to define how and where this functionality is applied without modifying the class to which we are applying the new feature. The cross-cutting concern can now be modularized into special classes, called **aspect**.

There are **two** benefits of aspects:

* First, the logic for each concern is now in one place instead of scattered all over the codebase.
* Second, the business modules only contain code for their primary concern. The secondary concern has been moved to the **aspect**.

The aspects have the responsibility that is to be implemented, called **advice**. We can implement an aspect's functionality into a program at one or more join points.

Benefits of AOP

* It is implemented in pure Java.
* There is no requirement for a special compilation process.
* It supports only method execution Join points.
* Only run time weaving is available.
* Two types of AOP proxy is available: **JDK dynamic proxy** and **CGLIB proxy.**

Cross-cutting concern

The cross-cutting concern is a concern that we want to implement in multiple places in an application. It affects the entire application.

AOP Terminology

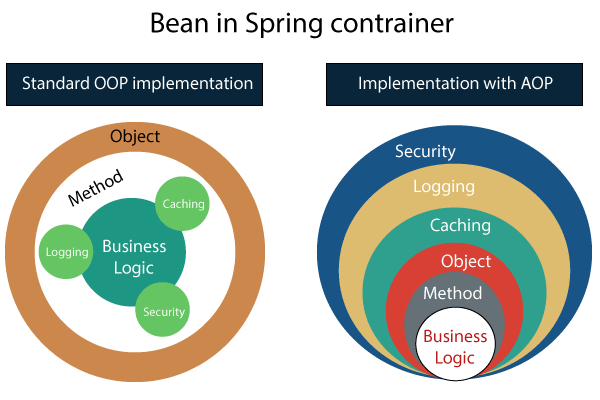
* **Aspect:** An aspect is a module that encapsulates **advice** and **pointcuts** and provides **cross-cutting** An application can have any number of aspects. We can implement an aspect using regular class annotated with **@Aspect** annotation.
* **Pointcut:** A pointcut is an expression that selects one or more join points where advice is executed. We can define pointcuts using **expressions** or **patterns**. It uses different kinds of expressions that matched with the join points. In Spring Framework, **AspectJ** pointcut expression language is used.
* **Join point:** A join point is a point in the application where we apply an **AOP aspect**. Or it is a specific execution instance of an advice. In AOP, join point can be a **method execution, exception handling, changing object variable value**, etc.
* **Advice:** The advice is an action that we take either **before** or **after** the method execution. The action is a piece of code that invokes during the program execution. There are **five** types of advices in the Spring AOP framework: **before, after, after-returning, after-throwing,**and **around advice.**Advices are taken for a particular **join point.**We will discuss these advices further in this section.
* **Target object:** An object on which advices are applied, is called the **target object**. Target objects are always a **proxied** It means a subclass is created at run time in which the target method is overridden, and advices are included based on their configuration.
* **Weaving:** It is a process of **linking aspects** with other application types. We can perform weaving at **run time, load time,** and **compile time**.

**Proxy:** It is an object that is created after applying advice to a target object is called **proxy**. The Spring AOP implements the **JDK dynamic proxy** to create the proxy classes with target classes and advice invocations. These are called AOP proxy classes.

AOP vs. OOP

The differences between AOP and OOP are as follows:

|  |  |
| --- | --- |
| **AOP** | **OOP** |
| **Aspect:** A code unit that encapsulates pointcuts, advices, and attributes. | **Class:** A code unit that encapsulates methods and attributes. |
| **Pointcut:** It defines the set of entry points in which advice is executed. | **Method signature:** It defines the entry points for the execution of method bodies. |
| **Advice:** It is an implementation of cross-cutting concerns. | **Method bodies:** It is an implementation of the business logic concerns. |
| **Waver: It constructs code (source or object) with advice.** | **Compiler: It converts source code to object code.** |



Spring AOP vs. AspectJ

The differences between AOP and OOP are as follows:

|  |  |
| --- | --- |
| **Spring AOP** | **AspectJ** |
| There is a need for a separate compilation process. | It requires the AspectJ compiler. |
| It supports only method execution pointcuts. | It supports all pointcuts. |
| It can be implemented on beans managed by Spring Container. | It can be implemented on all domain objects. |
| It supports only method level weaving. | It can wave fields, methods, constructors, static initializers, final class, etc. |

Types of AOP Advices

There are five types of AOP advices are as follows:

* Before Advice
* After Advice
* Around Advice
* After Throwing
* After Returning

**Before Advice:** An advice that executes before a join point, is called before advice. We use **@Before** annotation to mark an advice as Before advice.

**After Advice:** An advice that executes after a join point, is called after advice. We use **@After**annotation to mark an advice as After advice.

**Around Advice:** An advice that executes before and after of a join point, is called around advice.

**After Throwing Advice:** An advice that executes when a join point throws an exception.

**After Returning Advice:** An advice that executes when a method executes successfully.

Before implementing the AOP in an application, we are required to add **Spring AOP** dependency in the pom.xml file.

Spring Boot Starter AOP

Spring Boot Starter AOP is a dependency that provides Spring AOP and AspectJ. Where AOP provides basic AOP capabilities while the AspectJ provides a complete AOP framework.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **<version>**2.2.2.RELEASE**</version>**
5. **</dependency>**

In the next section, we will implement the different advices in the application.

Spring Boot AOP Before Advice

Before advice is used in Aspect-Oriented Programming to achieve the cross-cutting. It is an advice type which ensures that an advice runs before the method execution. We use **@Before** annotation to implement the before advice.

Let's understand before advice through an example.

Spring Boot Before Advice Example

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

**Step 2:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

14.1M

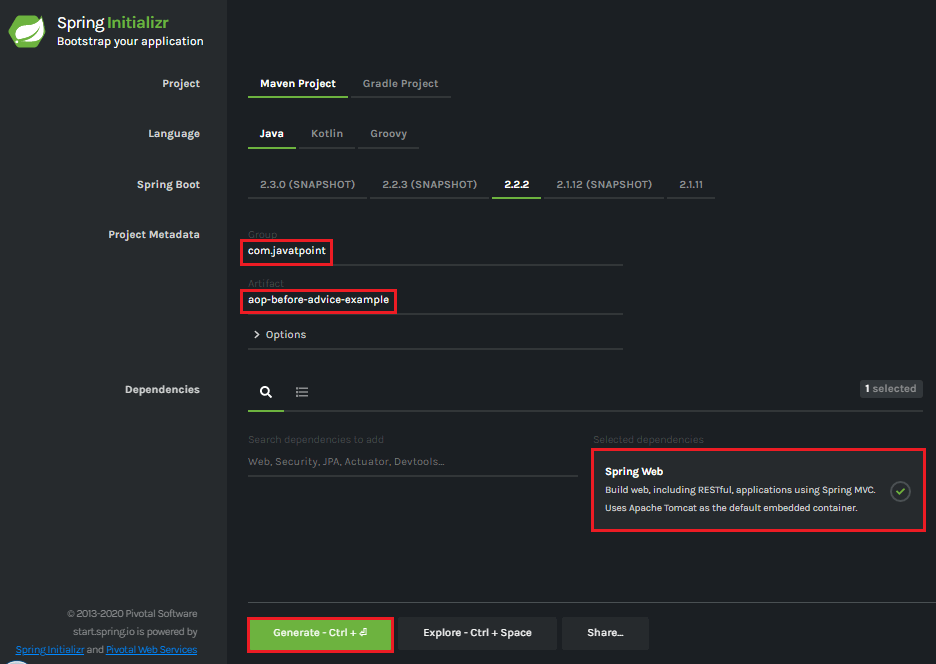
226

Triggers in SQL (Hindi)

**Step 3:** Provide the **Artifact Id.** We have provided the Artifact Id **aop-before-advice-example.**

**Step 4:** Add the **Spring Web**dependency.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications in a **jar** file and downloads it to the local system.



**Step 6: Extract** the downloaded jar file.

**Step 7: Import** the folder by using the following steps:

File -> Import -> Existing Maven Projects -> Next -> Browse the Folder **aop-before-advice-example**-> Finish.

**Step 8:** Open the **pom.xml**file and add the following **AOP** dependency. It is a starter for aspect-oriented programming with **Spring AOP** and **AspectJ**.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **</dependency>**
5. **</dependencies>**

**pom.xml**

1. **<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"**>**
2. **<modelVersion>**4.0.0**</modelVersion>**
3. **<groupId>**com.javatpoint**</groupId>**
4. **<artifactId>** aop-before-advice-example**</artifactId>**
5. **<version>**0.0.1-SNAPSHOT**</version>**
6. **<packaging>**jar**</packaging>**
7. **<name>**aop-before-advice-example**</name>**
8. **<description>**Demo project for Spring Boot**</description>**
9. **<parent>**
10. **<groupId>**org.springframework.boot**</groupId>**
11. **<artifactId>**spring-boot-starter-parent**</artifactId>**
12. **<version>**2.2.2.RELEASE**</version>**
13. **<relativePath** **/>** <!-- lookup parent from repository -->
14. **</parent>**
15. **<properties>**
16. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
17. **<project.reporting.outputEncoding>**UTF-8**</project.reporting.outputEncoding>**
18. **<java.version>**1.8**</java.version>**
19. **</properties>**
20. **<dependencies>**
21. **<dependency>**
22. **<groupId>**org.springframework.boot**</groupId>**
23. **<artifactId>**spring-boot-starter-web**</artifactId>**
24. **</dependency>**
25. **<dependency>**
26. **<groupId>**org.springframework.boot**</groupId>**
27. **<artifactId>**spring-boot-starter-aop**</artifactId>**
28. **</dependency>**
29. **</dependencies>**
31. **<build>**
32. **<plugins>**
33. **<plugin>**
34. **<groupId>**org.springframework.boot**</groupId>**
35. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
36. **</plugin>**
37. **</plugins>**
38. **</build>**
39. **</project>**

**Step 9:** Open **AopBeforeAdviceExampleApplication.java** file and add an annotation **@EnableAspectJAutoProxy.**

1. @EnableAspectJAutoProxy(proxyTargetClass=true)

It enables support for handling components marked with AspectJ’s @Aspect annotation. It is used with @Configuration annotation. We can control the type of proxy by using the **proxyTargetClass** attribute. Its default value is **false**.

**AopBeforeAdviceExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.context.annotation.EnableAspectJAutoProxy;
5. @SpringBootApplication
6. @EnableAspectJAutoProxy(proxyTargetClass=**true**)
7. **public** **class** AopBeforeAdviceExampleApplication
8. {
9. **public** **static** **void** main(String[] args) {
10. SpringApplication.run(AopBeforeAdviceExampleApplication.**class**, args);
11. }
12. }

**Step 10:** Create a package with the name **com.javatpoint.model.**

**Step 11:** Create a model class under the package **com.javatpoint.model.**We have created a class with the name **Employee.**In the class, define the following:

* Define three variables**empId, firstName,**and**secondName**of type String.
* Generate **Getters and Setters.**
* Create a **default**

**Employee.java**

1. **package** com.javatpoint.model;
2. **public** **class** Employee
3. {
4. **private** String empId;
5. **private** String firstName;
6. **private** String secondName;
7. //default constructor
8. **public** Employee()
9. {
10. }
11. **public** String getEmpId()
12. {
13. **return** empId;
14. }
15. **public** **void** setEmpId(String empId)
16. {
17. **this**.empId = empId;
18. }
19. **public** String getFirstName()
20. {
21. **return** firstName;
22. }
23. **public** **void** setFirstName(String firstName)
24. {
25. **this**.firstName = firstName;
26. }
27. **public** String getSecondName()
28. {
29. **return** secondName;
30. }
31. **public** **void** setSecondName(String secondName)
32. {
33. **this**.secondName = secondName;
34. }
35. }

**Step 12:** Create a package with the name **com.javatpoint.controller.**

**Step 13:** Create a controller class under the package **com.javatpoint.controller.**We have created a class with the name **EmployeeController.**

In the controller class, we have defined the two mappings one for adding an employee and the other for removing an employee.

**EmployeeController.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. **import** org.springframework.web.bind.annotation.RequestMethod;
5. **import** org.springframework.web.bind.annotation.RequestParam;
6. **import** org.springframework.web.bind.annotation.RestController;
7. **import** com.javatpoint.model.Employee;
8. **import** com.javatpoint.service.EmployeeService;
9. @RestController
10. **public** **class** EmployeeController
11. {
12. @Autowired
13. **private** EmployeeService employeeService;
14. @RequestMapping(value = "/add/employee", method = RequestMethod.GET)
15. **public** com.javatpoint.model.Employee addEmployee(@RequestParam("empId") String empId, @RequestParam("firstName") String firstName, @RequestParam("secondName") String secondName)
16. {
17. **return** employeeService.createEmployee(empId, firstName, secondName);
18. }
19. @RequestMapping(value = "/remove/employee", method = RequestMethod.GET)
20. **public** String removeEmployee( @RequestParam("empId") String empId)
21. {
22. employeeService.deleteEmployee(empId);
23. **return** "Employee removed";
24. }
25. }

**Step 14:** Create a package with the name **com.javatpoint.service.**

**Step 15:** Create a Service class under the package **com.javatpoint.service.**We have created a class with the name **EmployeeService.**

In the Service class, we have defined two methods **createEmployee**and**deleteEmployee.**

**EmployeeService.java**

1. **package** com.javatpoint.service;
2. **import** org.springframework.stereotype.Service;
3. **import** com.javatpoint.model.Employee;
4. @Service
5. **public** **class** EmployeeService
6. {
7. **public** Employee createEmployee( String empId, String fname, String sname)
8. {
9. Employee emp = **new** Employee();
10. emp.setEmpId(empId);
11. emp.setFirstName(fname);
12. emp.setSecondName(sname);
13. **return** emp;
14. }
15. **public** **void** deleteEmployee(String empId)
16. {
17. }
18. }

**Step 16:** Create a package with the name **com.javatpoint.aspect.**

**Step 17:** Create an aspect class under the package **com.javatpoint.aspect.**We have created a class with the name **EmployeeServiceAspect.**

In the aspect class, we have defined the before advice logic.

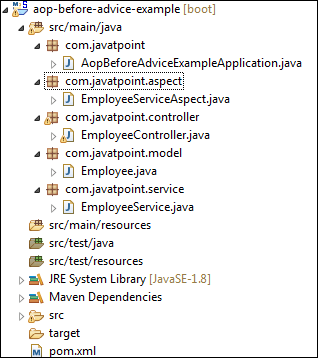
**EmployeeServiceAspect.java**

1. **package** com.javatpoint.aspect;
2. **import** org.aspectj.lang.JoinPoint;
3. **import** org.aspectj.lang.annotation.Aspect;
4. **import** org.aspectj.lang.annotation.Before;
5. **import** org.springframework.stereotype.Component;
6. @Aspect
7. @Component
8. **public** **class** EmployeeServiceAspect
9. {
10. @Before(value = "execution(\* com.javatpoint.service.EmployeeService.\*(..)) and args(empId, fname, sname)")
11. **public** **void** beforeAdvice(JoinPoint joinPoint, String empId, String fname, String sname) {
12. System.out.println("Before method:" + joinPoint.getSignature());
13. System.out.println("Creating Employee with first name - " + fname + ", second name - " + sname + " and id - " + empId);
14. }
15. }

In the above class:

* **execution(expression):** The expression is a method on which advice is to be applied.
* **@Before:** It marks a function as an advice to be executed before method that covered by PointCut.

After creating all the modules, the project directory looks like the following:

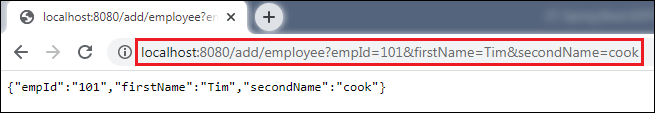


We have set-up all the modules. Now we will run the application.

**Step 18:** Open the e**AopBeforeAdviceExampleApplication.java** file and run it as Java Application.

**Step 19:** Open the browser and invoke the following URL : *http://localhost:8080/add/employee?empId={id}&firstName={fname}&secondName={sname}*

In the above URL, **/add/employee**is the mapping that we have created in the Controller class. We have used two separators **(?)**and **(&)**for separating two values.

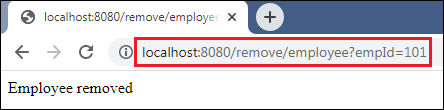


In the above output, we have assigned **emId 101, firstName=Tim,**and**secondName=cook.**

Let's have a look at the console. We see that before invoking the **createEmployee**() method of **EmployeeService** class, the method **beforeAdvice()** of **EmployeeServiceAspect** class invokes, as shown below.

Spring Boot AOP Before Advice

Similarly, we can also remove an employee by invoking the URL http://localhost:8080/remove/employee?empId=101. It returns a message **Employee removed**, as shown in the following figure.



In this section, we have learned the working of before advice. In the next section, we will learn the working of after advice and implement it in an apllication.

Spring Boot AOP After Advice

After advice is used in Aspect-Oriented Programming to achieve the cross-cutting. It is an advice type which ensures that an advice runs after the method execution. We use **@After** annotation to implement the after advice.

Let's understand after advice through an example.

Spring Boot After Advice Example

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

**Step 2:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

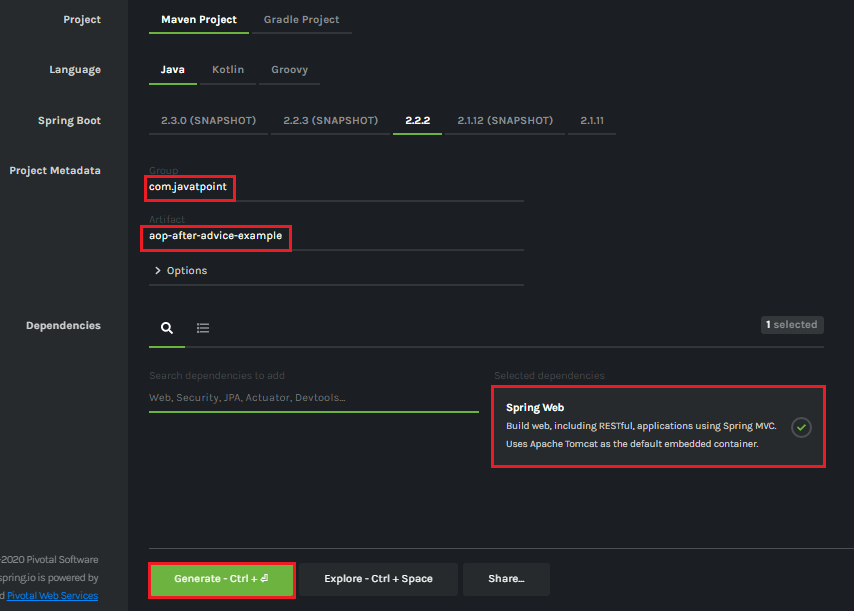
10 Sec

Hello Java Program for Beginners

**Step 3:** Provide the **Artifact Id.** We have provided the Artifact Id **aop-after-advice-example.**

**Step 4:** Add the **Spring Web** dependency.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications in a **jar** file and downloads it to the local system.



**Step 6: Extract** the downloaded jar file.

**Step 7: Import** the folder by using the following steps:

File -> Import -> Existing Maven Projects -> Next -> Browse the Folder **aop-after-advice-example**-> Finish.

**Step 8:** Open the **pom.xml**file and add the following **AOP** dependency. It is a starter for aspect-oriented programming with **Spring AOP** and **AspectJ**.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **</dependency>**
5. **</dependencies>**

**pom.xml**

1. **<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"**>**
2. **<modelVersion>**4.0.0**</modelVersion>**
3. **<groupId>**com.javatpoint**</groupId>**
4. **<artifactId>** aop-after-advice-example**</artifactId>**
5. **<version>**0.0.1-SNAPSHOT**</version>**
6. **<packaging>**jar**</packaging>**
7. **<name>**aop-after-advice-example**</name>**
8. **<description>**Demo project for Spring Boot**</description>**
9. **<parent>**
10. **<groupId>**org.springframework.boot**</groupId>**
11. **<artifactId>**spring-boot-starter-parent**</artifactId>**
12. **<version>**2.2.2.RELEASE**</version>**
13. **<relativePath** **/>** <!-- lookup parent from repository -->
14. **</parent>**
15. **<properties>**
16. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
17. **<project.reporting.outputEncoding>**UTF-8**</project.reporting.outputEncoding>**
18. **<java.version>**1.8**</java.version>**
19. **</properties>**
20. **<dependencies>**
21. **<dependency>**
22. **<groupId>**org.springframework.boot**</groupId>**
23. **<artifactId>**spring-boot-starter-web**</artifactId>**
24. **</dependency>**
25. **<dependency>**
26. **<groupId>**org.springframework.boot**</groupId>**
27. **<artifactId>**spring-boot-starter-aop**</artifactId>**
28. **</dependency>**
29. **</dependencies>**
31. **<build>**
32. **<plugins>**
33. **<plugin>**
34. **<groupId>**org.springframework.boot**</groupId>**
35. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
36. **</plugin>**
37. **</plugins>**
38. **</build>**
39. **</project>**

**Step 9:** Open **AopAfterAdviceExampleApplication.java** file and add an annotation **@EnableAspectJAutoProxy.**

1. @EnableAspectJAutoProxy(proxyTargetClass=true)

It enables support for handling components marked with AspectJ's **@Aspect** annotation. It is used with @Configuration annotation. We can control the type of proxy by using the **proxyTargetClass** attribute. Its default value is **false**.

**AopAfterAdviceExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.context.annotation.EnableAspectJAutoProxy;
5. @SpringBootApplication
6. @EnableAspectJAutoProxy(proxyTargetClass=**true**)
7. **public** **class** AopAfterAdviceExampleApplication
8. {
9. **public** **static** **void** main(String[] args) {
10. SpringApplication.run(AopAfterAdviceExampleApplication.**class**, args);
11. }
12. }

**Step 10:** Create a package with the name **com.javatpoint.model.**

**Step 11:** Create a model class under the package **com.javatpoint.model.**We have created a class with the name **Employee.**In the class, define the following:

* Define three variables**empId, firstName,**and**secondName**of type String.
* Generate **Getters and Setters.**
* Create a **default**

**Employee.java**

1. **package** com.javatpoint.model;
2. **public** **class** Employee
3. {
4. **private** String empId;
5. **private** String firstName;
6. **private** String secondName;
7. //default constructor
8. **public** Employee()
9. {
10. }
11. **public** String getEmpId()
12. {
13. **return** empId;
14. }
15. **public** **void** setEmpId(String empId)
16. {
17. **this**.empId = empId;
18. }
19. **public** String getFirstName()
20. {
21. **return** firstName;
22. }
23. **public** **void** setFirstName(String firstName)
24. {
25. **this**.firstName = firstName;
26. }
27. **public** String getSecondName()
28. {
29. **return** secondName;
30. }
31. **public** **void** setSecondName(String secondName)
32. {
33. **this**.secondName = secondName;
34. }
35. }

**Step 12:** Create a package with the name **com.javatpoint.controller.**

**Step 13:** Create a controller class under the package **com.javatpoint.controller.**We have created a class with the name **EmployeeController.**

In the controller class, we have defined the two mappings one for adding an employee and the other for removing an employee.

**EmployeeController.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. **import** org.springframework.web.bind.annotation.RequestMethod;
5. **import** org.springframework.web.bind.annotation.RequestParam;
6. **import** org.springframework.web.bind.annotation.RestController;
7. **import** com.javatpoint.model.Employee;
8. **import** com.javatpoint.service.EmployeeService;
9. @RestController
10. **public** **class** EmployeeController
11. {
12. @Autowired
13. **private** EmployeeService employeeService;
14. @RequestMapping(value = "/add/employee", method = RequestMethod.GET)
15. **public** com.javatpoint.model.Employee addEmployee(@RequestParam("empId") String empId, @RequestParam("firstName") String firstName, @RequestParam("secondName") String secondName)
16. {
17. **return** employeeService.createEmployee(empId, firstName, secondName);
18. }
19. @RequestMapping(value = "/remove/employee", method = RequestMethod.GET)
20. **public** String removeEmployee( @RequestParam("empId") String empId)
21. {
22. employeeService.deleteEmployee(empId);
23. **return** "Employee removed";
24. }
25. }

**Step 14:** Create a package with the name **com.javatpoint.service.**

**Step 15:** Create a Service class under the package **com.javatpoint.service.**We have created a class with the name **EmployeeService.**

In the Service class, we have defined two methods **createEmployee**and**deleteEmployee.**

**EmployeeService.java**

1. **package** com.javatpoint.service;
2. **import** org.springframework.stereotype.Service;
3. **import** com.javatpoint.model.Employee;
4. @Service
5. **public** **class** EmployeeService
6. {
7. **public** Employee createEmployee( String empId, String fname, String sname)
8. {
9. Employee emp = **new** Employee();
10. emp.setEmpId(empId);
11. emp.setFirstName(fname);
12. emp.setSecondName(sname);
13. **return** emp;
14. }
15. **public** **void** deleteEmployee(String empId)
16. {
17. }
18. }

**Step 16:** Create a package with the name **com.javatpoint.aspect.**

**Step 17:** Create an aspect class under the package **com.javatpoint.aspect.**We have created a class with the name **EmployeeServiceAspect.**

In the aspect class, we have defined the after-advice logic.

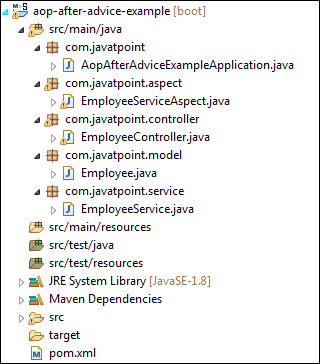
**EmployeeServiceAspect.java**

1. package com.javatpoint.aspect;
2. import org.aspectj.lang.JoinPoint;
3. import org.aspectj.lang.annotation.Aspect;
4. import org.aspectj.lang.annotation.After;
5. import org.springframework.stereotype.Component;
6. @Aspect
7. @Component
8. public class EmployeeServiceAspect
9. {
10. @After(value = "execution(\* com.javatpoint.service.EmployeeService.\*(..)) and args(empId, fname, sname)")
11. public void afterAdvice(JoinPoint joinPoint, String empId, String fname, String sname) {
12. System.out.println("After method:" + joinPoint.getSignature());
13. System.out.println("Creating Employee with first name - " + fname + ", second name - " + sname + " and id - " + empId);
14. }
15. }

In the above class:

* **execution(expression):** The expression is a method on which advice is to be applied.
* **@After:** The method annotated with **@After** executes after all the methods that matched with the pointcut expression.

After creating all the modules, the project directory looks like the following:

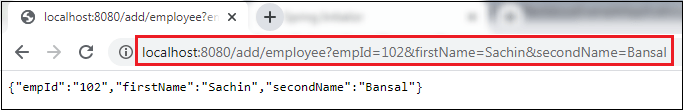


We have set-up all the modules. Now we will run the application.

**Step 18:** Open the **AopAfterAdviceExampleApplication.java** file and run it as Java Application.

**Step 19:** Open the browser and invoke the following URL : *http://localhost:8080/add/employee?empId={id}&firstName={fname}&secondName={sname}*

In the above URL, **/add/employee**is the mapping that we have created in the Controller class. We have used two separators **(?)**and **(&)**for separating two values.

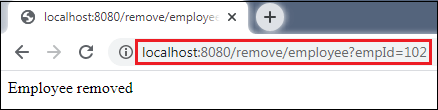


In the above output, we have assigned **emId 102, firstName=Sachin,**and**secondName=Bansal.**

Let's have a look at the console. We see that after invoking the **createEmployee**() method of **EmployeeService** class, the method **afterAdvice()** of **EmployeeServiceAspect** class invokes, as shown below.

Spring Boot AOP After Advice

Similarly, we can also remove an employee by invoking the URL http://localhost:8080/remove/employee?empId=102. It returns a message **Employee removed**, as shown in the following figure.



In this section, we have learned the working of after advice. In the next section, we will learn the working of around advice.

Spring Boot AOP Around Advice

Around advice is represented by **@Around** annotation. It executes before and after a join point. It is the most powerful advice. It also provides more control for end-user to get deal with **ProceedingJoinPoint.**

Let's implement around advice in an application.

Spring Boot Around Advice Example

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/" \t "_blank)

.

**Step 2:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

25.2M

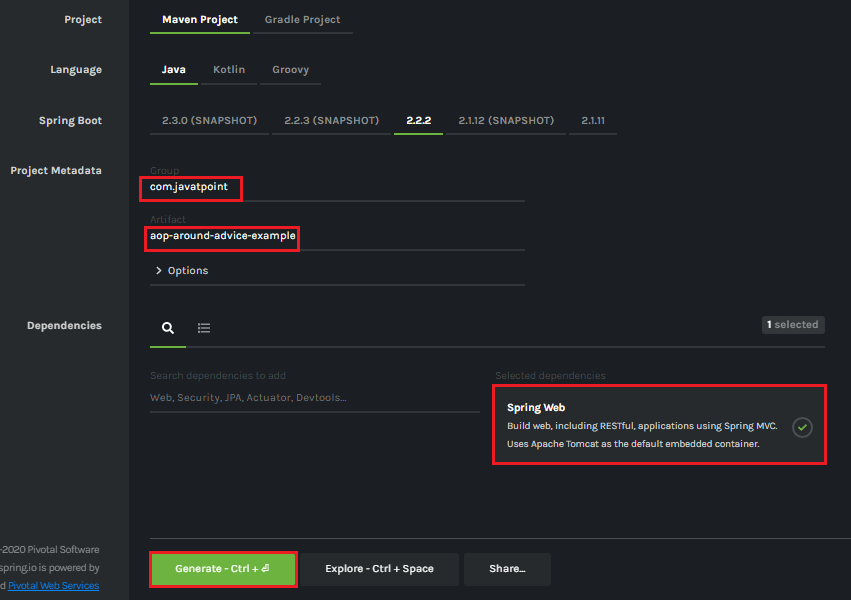
524

HTML Tutorial

**Step 3:** Provide the **Artifact Id.** We have provided the Artifact Id **aop-around-advice-example.**

**Step 4:** Add the **Spring Web** dependency.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications in a **jar** file and downloads it to the local system.



**Step 6: Extract** the downloaded jar file.

**Step 7: Import** the folder by using the following steps:

File -> Import -> Existing Maven Projects -> Next -> Browse the Folder **aop-around-advice-example**-> Finish.

**Step 8:** Open the **pom.xml**file and add the following **AOP** dependency. It is a starter for aspect-oriented programming with **Spring AOP** and **AspectJ**.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **</dependency>**
5. **</dependencies>**

**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" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**org.springframework.boot**</groupId>**
6. **<artifactId>**spring-boot-starter-parent**</artifactId>**
7. **<version>**2.2.2.RELEASE**</version>**
8. **<relativePath/>** <!-- lookup parent from repository -->
9. **</parent>**
10. **<groupId>**com.javatpoint**</groupId>**
11. **<artifactId>**aop-around-advice-example**</artifactId>**
12. **<version>**0.0.1-SNAPSHOT**</version>**
13. **<name>**aop-around-advice-example**</name>**
14. **<description>**Demo project for Spring Boot**</description>**
15. **<properties>**
16. **<java.version>**1.8**</java.version>**
17. **</properties>**
18. **<dependencies>**
19. **<dependency>**
20. **<groupId>**org.springframework.boot**</groupId>**
21. **<artifactId>**spring-boot-starter-web**</artifactId>**
22. **</dependency>**
23. **<dependency>**
24. **<groupId>**org.springframework.boot**</groupId>**
25. **<artifactId>**spring-boot-starter-aop**</artifactId>**
26. **</dependency>**
27. **<dependency>**
28. **<groupId>**org.springframework.boot**</groupId>**
29. **<artifactId>**spring-boot-starter-test**</artifactId>**
30. **<scope>**test**</scope>**
31. **<exclusions>**
32. **<exclusion>**
33. **<groupId>**org.junit.vintage**</groupId>**
34. **<artifactId>**junit-vintage-engine**</artifactId>**
35. **</exclusion>**
36. **</exclusions>**
37. **</dependency>**
38. **</dependencies>**
39. **<build>**
40. **<plugins>**
41. **<plugin>**
42. **<groupId>**org.springframework.boot**</groupId>**
43. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
44. **</plugin>**
45. **</plugins>**
46. **</build>**
47. **</project>**

**Step 9:** Create a package with the name **com.javatpoint.service.**

**Step 10:** Create a class in the above package with the name **BankService**.

In this class, we have defined a method named **displayBalance().**It checks the account number. If the account number is matched returns total amount, else returns a message.

**BankService.java**

1. **package** com.javatpoint.service;
2. **import** org.springframework.stereotype.Service;
3. @Service
4. **public** **class** BankService
5. {
6. **public** **void** displayBalance(String accNum)
7. {
8. System.out.println("Inside displayBalance() method");
9. **if**(accNum.equals("12345"))
10. {
11. System.out.println("Total balance: 10,000");
12. }
13. **else**
14. {
15. System.out.println("Sorry! wrong account number.");
16. }
17. }
18. }

**Step 11:**Create another package with the name **com.javatpoint.aspect.**

**Step 12:**Create a class in the above package with the name **BankAspect.**

In the following class, we have defined two methods named **logDisplayingBalance()** and **aroundAdvice()** method.

**BankAspect.java**

1. **package** com.javatpoint.aspect;
2. **import** org.aspectj.lang.ProceedingJoinPoint;
3. **import** org.aspectj.lang.annotation.Around;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.aspectj.lang.annotation.Pointcut;
6. **import** org.springframework.stereotype.Component;
7. //Enables the spring AOP functionality in an application
8. @Aspect
9. @Component
10. **public** **class** BankAspect
11. {
12. //Displays all the available methods i.e. the advice will be called for all the methods
13. @Pointcut(value= "execution(\* com.javatpoint.service.BankService.\*(..))")
14. **private** **void** logDisplayingBalance()
15. {
16. }
17. //Declares the around advice that is applied before and after the method matching with a pointcut expression
18. @Around(value= "logDisplayingBalance()")
19. **public** **void** aroundAdvice(ProceedingJoinPoint jp) **throws** Throwable
20. {
21. System.out.println("The method aroundAdvice() before invokation of the method " + jp.getSignature().getName() + " method");
22. **try**
23. {
24. jp.proceed();
25. }
26. **finally**
27. {
29. }
30. System.out.println("The method aroundAdvice() after invokation of the method " + jp.getSignature().getName() + " method");
31. }
32. }

**Step 13:** Open **AopAroundAdviceExampleApplication.java** file and add an annotation **@EnableAspectJAutoProxy.**

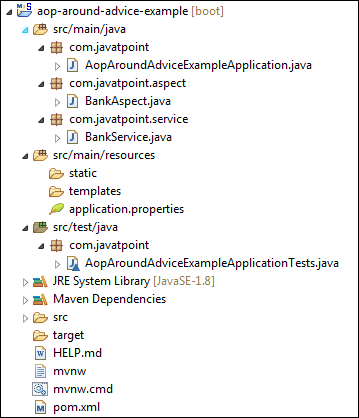
The annotation enables support for handling components marked with AspectJ's **@Aspect** annotation. It is used with @Configuration annotation.

**ConfigurableApplicationContext** is an interface that provides facilities to configure an application context in addition to the application context client methods in the ApplicationContext.

**AopAroundAdviceExampleApplication.java**

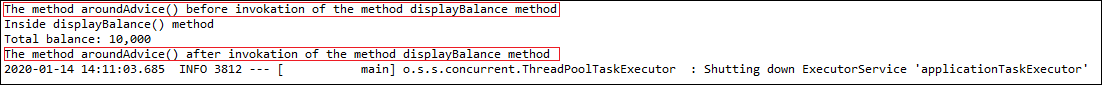
1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.context.ConfigurableApplicationContext;
5. **import** org.springframework.context.annotation.EnableAspectJAutoProxy;
6. **import** com.javatpoint.service.BankService;
7. @SpringBootApplication
8. //@EnableAspectJAutoProxy annotation enables support for handling the components marked with @Aspect annotation. It is similar to tag in the xml configuration.
9. @EnableAspectJAutoProxy
10. **public** **class** AopAroundAdviceExampleApplication
11. {
12. **public** **static** **void** main(String[] args)
13. {
14. ConfigurableApplicationContext context = SpringApplication.run(AopAroundAdviceExampleApplication.**class**, args);
15. // Fetching the employee object from the application context.
16. BankService bank = context.getBean(BankService.**class**);
17. // Displaying balance in the account
18. String accnumber = "12345";
19. bank.displayBalance(accnumber);
20. // Closing the context object
21. context.close();
22. }
23. }

After creating all the packages and classes the project directory looks like the following:



Now, run the application.

**Step 14:** Open the **AopAroundAdviceExampleApplication.java** and run it as Java Application.



In the above output, we see that the method aroundAdvice() invokes two times. First, before the execution of **the displayBalance()** method and second, after the execution of the **displayBalance()** method. It is called around advice.

Spring Boot AOP After Returning Advice

**After returning** is an advice in Spring AOP that invokes after the execution of join point complete (execute) normally. It does not invoke if an exception is thrown. We can implement after returning advice in an application by using **@AfterReturning** annotation. The annotation marks a function as an advice to be executed before the method covered by PointCut.

After returning advice runs when a matched method execution returns a value normally. The name that we define in the return attribute must correspond to the name of a parameter in the advice method. When a method returns a value, the value will be passed to the advice method as the corresponding argument value.

Let's implement the after returning advice in an application.

Spring Boot After Returning Advice Example

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/" \t "_blank)

.

25.2M

524

HTML Tutorial

**Step 2:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

**Step 3:** Provide the **Artifact Id.** We have provided the Artifact Id **aop-after-returning-advice-example.**

**Step 4:** Add the **Spring Web** dependency.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications in a **jar** file and downloads it to the local system.

**Step 6: Extract** the downloaded jar file.

**Step 7: Import** the folder by using the following steps:

File -> Import -> Existing Maven Projects -> Next -> Browse the Folder **aop-after-returning-advice-example**-> Finish.

**Step 8:** Open the **pom.xml**file and add the following **AOP** dependency. It is a starter for aspect-oriented programming with **Spring AOP** and **AspectJ**.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **</dependency>**
5. **</dependencies>**
6. pom.xml
7. **<?xml** version="1.0" encoding="UTF-8"**?>**
8. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
9. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
10. **<modelVersion>**4.0.0**</modelVersion>**
11. **<groupId>**com.javatpoint**</groupId>**
12. **<artifactId>**aop-after-returning-advice-example**</artifactId>**
13. **<version>**0.0.1-SNAPSHOT**</version>**
14. **<packaging>**jar**</packaging>**
15. **<name>**aop-after-returning-advice-example**</name>**
16. **<description>**Demo project for Spring Boot**</description>**
17. **<parent>**
18. **<groupId>**org.springframework.boot**</groupId>**
19. **<artifactId>**spring-boot-starter-parent**</artifactId>**
20. **<version>**2.2.2.RELEASE**</version>**
21. **<relativePath/>** <!-- lookup parent from repository -->
22. **</parent>**
23. **<properties>**
24. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
25. **<project.reporting.outputEncoding>**UTF-8**</project.reporting.outputEncoding>**
26. **<java.version>**1.8**</java.version>**
27. **</properties>**
28. **<dependencies>**
29. **<dependency>**
30. **<groupId>**org.springframework.boot**</groupId>**
31. **<artifactId>**spring-boot-starter-aop**</artifactId>**
32. **</dependency>**
33. **<dependency>**
34. **<groupId>**org.springframework.boot**</groupId>**
35. **<artifactId>**spring-boot-starter-test**</artifactId>**
36. **<scope>**test**</scope>**
37. **</dependency>**
38. **</dependencies>**
39. **<build>**
40. **<plugins>**
41. **<plugin>**
42. **<groupId>**org.springframework.boot**</groupId>**
43. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
44. **</plugin>**
45. **</plugins>**
46. **</build>**
47. **</project>**

**Step 9:** Create a package with the name**com.javatpoint.model**in**src/main/java**folder**.**

**Step 10:** Create a class with the name **Account**in the package **com.javatpoint.model.**

In the Account class, do the following:

* Defined two variables **accountNumber** and **accountType** of type String.
* Right-click on the file -> Source -> Generate Constructor using Fields
* Generate Getters.  
  Right-click on the file -> Source -> Generate Getters and Setters -> Select Getters -> Generate
* Generate a **toString()**  
  Right-click on the file -> Source -> Generate toString()…

**Account.java**

1. **package** com.javatpoint.model;
2. **public** **class** Account
3. {
4. **private** String accountNumber;
5. **private** String accountType;
6. **public** Account(String accountNumber, String accountType)
7. {
8. **super**();
9. **this**.accountNumber = accountNumber;
10. **this**.accountType = accountType;
11. }
12. **public** String getAccountType()
13. {
14. **return** accountType;
15. }
16. **public** String getAccountNumber()
17. {
18. **return** accountNumber;
19. }
20. @Override
21. **public** String toString()
22. {
23. **return** "Account [accountNumber=" + accountNumber+ ", accountType=" + accountType + "]";
24. }
25. }

**Step 11:** Create another package with the name **com.javatpoint.service.impl.**

**Step 12:** In this package, create a class with the name **AccountServiceImple.**

In this class, we have defined account service.

**AccountServiceImpl.java**

1. **package** com.javatpoint.service.impl;
2. **import** java.util.HashMap;
3. **import** java.util.Map;
4. **import** java.util.Map.Entry;
5. **import** java.util.Set;
6. **import** org.springframework.stereotype.Service;
7. **import** com.javatpoint.model.Account;
8. @Service
9. **public** **class** AccountServiceImpl **implements** AccountService
10. {
11. //storing account detail in the HashMap
12. **private** **static** Map<String,Account> map = **null**;
13. **static**
14. {
15. map = **new** HashMap<>();
16. //adding account detail in the map
17. map.put("M4546779", **new** Account("10441117000", "Saving Account"));
18. map.put("K2434567", **new** Account("10863554577", "Current Account"));
19. }
20. @Override
21. **public** Account getAccountByCustomerId(String customerId) **throws** Exception
22. {
23. **if**(customerId ==**null**)
24. {
25. **throw** **new** Exception("Invalid! Customer Id");
26. }
27. Account account= **null**;
28. Set<Entry<String, Account>> entrySet = map.entrySet();
29. **for** (Entry<String, Account> entry : entrySet)
30. {
31. **if**(entry.getKey().equals(customerId))
32. {
33. account= entry.getValue();
34. }
35. }
36. **return** account;
37. }
38. }

**Step 13:** Create an interface with the name **AccountService** in the package **com.javatpoint.service.impl.**

**AccountService.java**

1. **package** com.javatpoint.service.impl;
2. **import** com.javatpoint.model.Account;
3. //creating interface that throws exception if the customer id not found
4. **public** **interface** AccountService
5. {
6. **public** **abstract** Account getAccountByCustomerId(String customerId)
7. **throws** Exception;
8. }

**Step 14:** Create a package with the name **com.javatpoint.aspect.**

**Step 15:** Create a class with the name **AccountAspect**in the package **com.javatpoint.aspect**.

In this class, we have implemented the after returning advice by using the annotation **@AfterReturning.**We have also defined a method **afterReturningAdvice()** method.

**Note:** The name (account) that we define in the **returning** attribute must correspond to the name of a parameter in the **advice method**.

**AccountAspect.java**

1. **package** com.javatpoint.aspect;
2. **import** org.aspectj.lang.JoinPoint;
3. **import** org.aspectj.lang.annotation.AfterReturning;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.springframework.stereotype.Component;
6. **import** com.javatpoint.model.Account;
7. @Aspect
8. @Component
9. **public** **class** AccountAspect
10. {
11. //implementing after returning advice
12. @AfterReturning(value="execution(\* com.javatpoint.service.impl.AccountServiceImpl.\*(..))",returning="account")
13. **public** **void** afterReturningAdvice(JoinPoint joinPoint, Account account)
14. {
15. System.out.println("After Returing method:"+joinPoint.getSignature());
16. System.out.println(account);
17. }
18. }

**Step 16:** Open the **AopAfterReturningAdviceExampleApplication.java** file and add an annotation **@EnableAspectJAutoProxy.**

The annotation enables support for handling components marked with AspectJ's **@Aspect** annotation. It is used with @Configuration annotation.

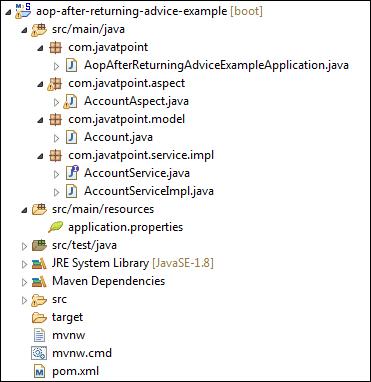
We have used the **proxyTargetClass** attribute of the annotation @EnableAspectJAutoProxy. The attribute **proxyTargetClass=true**allows us to use **CGLIB** (Code Generation Library) proxies instead of the default interface-based JDK proxy approach.

**ConfigurableApplicationContext**is an interface that provides facilities to configure an application context in addition to the application context client methods in the ApplicationContext.

**AopAfterReturningAdviceExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.context.ConfigurableApplicationContext;
5. **import** org.springframework.context.annotation.EnableAspectJAutoProxy;
6. **import** com.javatpoint.model.Account;
7. **import** com.javatpoint.service.impl.AccountService;
8. **import** com.javatpoint.service.impl.AccountServiceImpl;
9. @SpringBootApplication
10. //@EnableAspectJAutoProxy annotation enables support for handling the components marked with @Aspect annotation. It is similar to tag in the xml configuration.
11. @EnableAspectJAutoProxy(proxyTargetClass=**true**)
12. **public** **class** AopAfterReturningAdviceExampleApplication
13. {
14. **public** **static** **void** main(String[] args)
15. {
16. ConfigurableApplicationContext ac = SpringApplication.run(AopAfterReturningAdviceExampleApplication.**class**, args);
17. //Fetching the account object from the application context
18. AccountService accountService = ac.getBean("accountServiceImpl", AccountServiceImpl.**class**);
19. Account account;
20. **try**
21. {
22. account = accountService.getAccountByCustomerId("K2434567");
23. **if**(account != **null**)
24. System.out.println(account.getAccountNumber()+"\t"+account.getAccountType());
25. }
26. **catch** (Exception e)
27. {
28. System.out.println(e.getMessage());
29. }
30. }
31. }

After creating all the class and packages, the project directory looks like the following:



**Step 17:** Open the **AopAfterReturningAdviceExampleApplication.java** file and run it as Java Application. It shows the output, as shown below:

Spring Boot AOP After Returning Advice

In the next section, we will understand after throwing advice.

# Spring Boot AOP After Throwing Advice

After throwing is an advice type in Spring AOP. It ensures that an advice runs if a method throws an exception. We use **@AfterThrowing** annotation to implement the after throwing advice.

**Syntax:**

1. @AfterThrowing(PointCut="execution(expression) ", throwing="name")

Where:

**PointCut:** It selects a function.

32.4M

706

Exception Handling in Java - Javatpoint

**execution(expression):** It is an expression on which advice is to be applied.

**throwing:** The name of the exception to be returned.

Let's implement the after throwing advice in an application.

## Spring Boot After Throwing Advice Example

We will use the previous example in this section. You can download the project or make some modifications in the previous example.

## Spring Boot After Throwing Advice Example

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

**Step 2:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

**Step 3:** Provide the **Artifact Id.** We have provided the Artifact Id **aop-after-throwing-advice-example.**

**Step 4:** Add the **Spring Web** dependency.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications in a **jar** file and downloads it to the local system.

**Step 6: Extract** the downloaded jar file.

**Step 7:** **Import** the folder by using the following steps:

File -> Import -> Existing Maven Projects -> Next -> Browse the Folder **aop-after-throwing-advice-example**-> Finish.

**Step 8:** Open the **pom.xml**file and add the following **AOP** dependency. It is a starter for aspect-oriented programming with **Spring AOP** and **AspectJ**.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **</dependency>**
5. **</dependencies>**

**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. **<groupId>**com.javatpoint**</groupId>**
6. **<artifactId>**aop-after-throwing-advice-example**</artifactId>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **<packaging>**jar**</packaging>**
9. **<name>**aop-after-throwing-advice-example**</name>**
10. **<description>**Demo project for Spring Boot**</description>**
11. **<parent>**
12. **<groupId>**org.springframework.boot**</groupId>**
13. **<artifactId>**spring-boot-starter-parent**</artifactId>**
14. **<version>**2.2.2.RELEASE**</version>**
15. **<relativePath/>** <!-- lookup parent from repository -->
16. **</parent>**
17. **<properties>**
18. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
19. **<project.reporting.outputEncoding>**UTF-8**</project.reporting.outputEncoding>**
20. **<java.version>**1.8**</java.version>**
21. **</properties>**
22. **<dependencies>**
23. **<dependency>**
24. **<groupId>**org.springframework.boot**</groupId>**
25. **<artifactId>**spring-boot-starter-aop**</artifactId>**
26. **</dependency>**
27. **<dependency>**
28. **<groupId>**org.springframework.boot**</groupId>**
29. **<artifactId>**spring-boot-starter-test**</artifactId>**
30. **<scope>**test**</scope>**
31. **</dependency>**
32. **</dependencies>**
33. **<build>**
34. **<plugins>**
35. **<plugin>**
36. **<groupId>**org.springframework.boot**</groupId>**
37. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
38. **</plugin>**
39. **</plugins>**
40. **</build>**
41. **</project>**

**Step 9:** Create a package with the name**com.javatpoint.model**in**src/main/java**folder**.**

**Step 10:** Create a class with the name **Account**in the package **com.javatpoint.model.**

In the Account class, do the following:

* Defined two variables **accountNumber** and **accountType** of type String.
* Right-click on the file -> Source -> Generate Constructor using Fields
* Generate Getters.  
  Right-click on the file -> Source -> Generate Getters and Setters -> Select Getters -> Generate
* Generate a **toString()**  
  Right-click on the file -> Source -> Generate toString()

**Account.java**

1. **package** com.javatpoint.model;
2. **public** **class** Account
3. {
4. **private** String accountNumber;
5. **private** String accountType;
6. **public** Account(String accountNumber, String accountType)
7. {
8. **super**();
9. **this**.accountNumber = accountNumber;
10. **this**.accountType = accountType;
11. }
12. **public** String getAccountType()
13. {
14. **return** accountType;
15. }
16. **public** String getAccountNumber()
17. {
18. **return** accountNumber;
19. }
20. @Override
21. **public** String toString()
22. {
23. **return** "Account [accountNumber=" + accountNumber+ ", accountType=" + accountType + "]";
24. }
25. }

**Step 11:** Create another package with the name **com.javatpoint.service.impl.**

**Step 12:** In this package, create a class with the name **AccountServiceImple.**

In this class, we have defined account service.

**AccountServiceImpl.java**

1. **package** com.javatpoint.service.impl;
2. **import** java.util.HashMap;
3. **import** java.util.Map;
4. **import** java.util.Map.Entry;
5. **import** java.util.Set;
6. **import** org.springframework.stereotype.Service;
7. **import** com.javatpoint.model.Account;
8. @Service
9. **public** **class** AccountServiceImpl **implements** AccountService
10. {
11. //storing account detail in the HashMap
12. **private** **static** Map<String,Account> map = **null**;
13. **static**
14. {
15. map = **new** HashMap<>();
16. //adding account detail in the map
17. map.put("M4546779", **new** Account("10441117000", "Saving Account"));
18. map.put("K2434567", **new** Account("10863554577", "Current Account"));
19. }
20. @Override
21. **public** Account getAccountByCustomerId(String customerId) **throws** Exception
22. {
23. **if**(customerId ==**null**)
24. {
25. **throw** **new** Exception("Invalid! Customer Id");
26. }
27. Account account= **null**;
28. Set<Entry<String, Account>> entrySet = map.entrySet();
29. **for** (Entry<String, Account> entry : entrySet)
30. {
31. **if**(entry.getKey().equals(customerId))
32. {
33. account= entry.getValue();
34. }
35. }
36. **return** account;
37. }
38. }

**Step 13:** Create an interface with the name **AccountService** in the package **com.javatpoint.service.impl.**

**AccountService.java**

1. **package** com.javatpoint.service.impl;
2. **import** com.javatpoint.model.Account;
3. //creating interface that throws exception if the customer id not found
4. **public** **interface** AccountService
5. {
6. **public** **abstract** Account getAccountByCustomerId(String customerId)
7. **throws** Exception;
8. }

**Step 14:** Create a package with the name **com.javatpoint.aspect.**

**Step 15:** Create a class with the name **AccountAspect**in the package **com.javatpoint.aspect**.

In this class, we have implemented the after throwing advice by using the annotation **@AfterThrowing.**We have also defined a method **afterThrowingAdvice()** method.

#### **Note: The name (ex) that we define in the throwing attribute must correspond to the name of a parameter in the advice method. Otherwise, advice will not run.**

**AccountAspect.java**

1. **package** com.javatpoint.aspect;
2. **import** org.aspectj.lang.JoinPoint;
3. **import** org.aspectj.lang.annotation.AfterThrowing;
4. **import** org.aspectj.lang.annotation.Aspect;
5. **import** org.springframework.stereotype.Component;
6. @Aspect
7. @Component
8. **public** **class** AccountAspect
9. {
10. //implementing after throwing advice
11. @AfterThrowing(value="execution(\* com.javatpoint.service.impl.AccountServiceImpl.\*(..))",throwing="ex")
12. **public** **void** afterThrowingAdvice(JoinPoint joinPoint, Exception ex)
13. {
14. System.out.println("After Throwing exception in method:"+joinPoint.getSignature());
15. System.out.println("Exception is:"+ex.getMessage());
16. }
17. }

**Step 16:** Open the **AopAfterThrowingAdviceExampleApplication.java** file and add an annotation **@EnableAspectJAutoProxy.**

The annotation enables support for handling components marked with AspectJ's **@Aspect** annotation. It is used with @Configuration annotation.

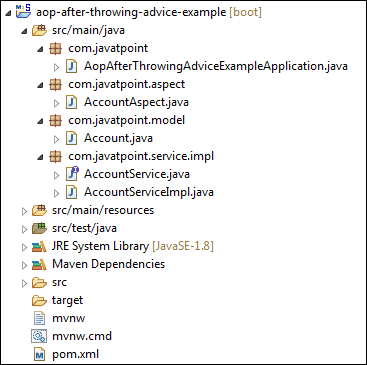
We have used the **proxyTargetClass** attribute of the annotation @EnableAspectJAutoProxy. The attribute **proxyTargetClass=true**allows us to use **CGLIB** (Code Generation Library) proxies instead of the default interface-based JDK proxy approach.

**ConfigurableApplicationContext**is an interface that provides facilities to configure an application context in addition to the application context client methods in the ApplicationContext.

**AopAfterThrowingAdviceExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.context.ConfigurableApplicationContext;
5. **import** org.springframework.context.annotation.EnableAspectJAutoProxy;
6. **import** com.javatpoint.model.Account;
7. **import** com.javatpoint.service.impl.AccountService;
8. **import** com.javatpoint.service.impl.AccountServiceImpl;
9. @SpringBootApplication
10. //@EnableAspectJAutoProxy annotation enables support for handling the components marked with @Aspect annotation. It is similar to tag in the xml configuration.
11. @EnableAspectJAutoProxy(proxyTargetClass=**true**)
12. **public** **class** AopAfterThrowingAdviceExampleApplication
13. {
14. **public** **static** **void** main(String[] args)
15. {
16. ConfigurableApplicationContext ac = SpringApplication.run(AopAfterThrowingAdviceExampleApplication.**class**, args);
17. //Fetching the account object from the application context
18. AccountService accountService = ac.getBean("accountServiceImpl", AccountServiceImpl.**class**);
19. Account account;
20. **try**
21. {
22. //generating exception
23. account = accountService.getAccountByCustomerId(**null**);
24. **if**(account != **null**)
25. System.out.println(account.getAccountNumber()+"\t"+account.getAccountType());
26. }
27. **catch** (Exception e)
28. {
29. System.out.println(e.getMessage());
30. e.printStackTrace();
31. }
32. }
33. }

After creating all the class and packages, the project directory looks like the following:



**Step 17:** Open the **AopAfterThrowingAdviceExampleApplication.java** file and run it as Java Application. It shows the output, as shown below:

Spring Boot AOP After Throwing Advice

Spring Boot JPA

What is JPA?

**Spring Boot JPA**is a Java specification for managing **relational** data in Java applications. It allows us to access and persist data between Java object/ class and relational database. JPA follows **Object-Relation Mapping**(ORM). It is a set of interfaces. It also provides a runtime **EntityManager** API for processing queries and transactions on the objects against the database. It uses a platform-independent object-oriented query language JPQL (Java Persistent Query Language).

In the context of persistence, it covers three areas:

* The Java Persistence API
* **Object-Relational** metadata
* The API itself, defined in the **persistence** package

JPA is not a framework. It defines a concept that can be implemented by any framework.

Why should we use JPA?

JPA is simpler, cleaner, and less labor-intensive than JDBC, SQL, and hand-written mapping. JPA is suitable for non-performance oriented complex applications. The main advantage of JPA over JDBC is that, in JPA, data is represented by objects and classes while in JDBC data is represented by tables and records. It uses POJO to represent persistent data that simplifies database programming. There are some other advantages of JPA:

31.7M

733

Features of Java - Javatpoint

* JPA avoids writing DDL in a database-specific dialect of SQL. Instead of this, it allows mapping in XML or using Java annotations.
* JPA allows us to avoid writing DML in the database-specific dialect of SQL.
* JPA allows us to save and load Java objects and graphs without any DML language at all.
* When we need to perform queries JPQL, it allows us to express the queries in terms of Java entities rather than the (native) SQL table and columns.

JPA Features

There are following features of JPA:

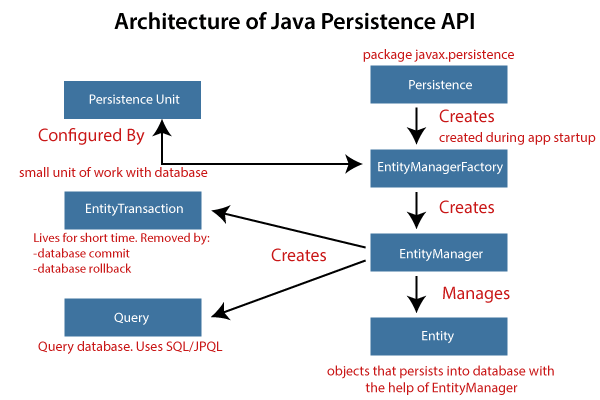
* It is a powerful repository and custom **object-mapping abstraction.**
* It supports for **cross-store persistence**. It means an entity can be partially stored in MySQL and Neo4j (Graph Database Management System).
* It dynamically generates queries from queries methods name.
* The domain base classes provide basic properties.
* It supports transparent auditing.
* Possibility to integrate custom repository code.
* It is easy to integrate with Spring Framework with the custom namespace.

JPA Architecture

JPA is a source to store business entities as relational entities. It shows how to define a POJO as an entity and how to manage entities with relation.

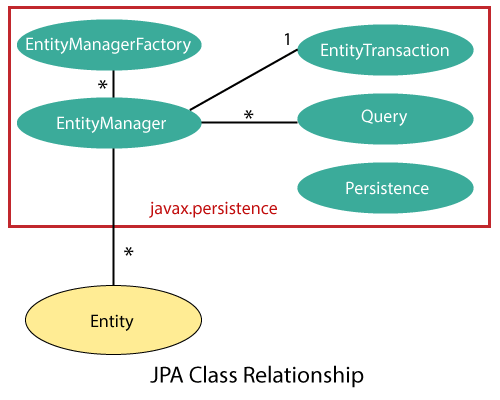
The following figure describes the class-level architecture of JPA that describes the core classes and interfaces of JPA that is defined in the **javax persistence** package. The JPA architecture contains the following units:

* **Persistence:** It is a class that contains static methods to obtain an EntityManagerFactory instance.
* **EntityManagerFactory:** It is a factory class of EntityManager. It creates and manages multiple instances of EntityManager.
* **EntityManager:** It is an interface. It controls the persistence operations on objects. It works for the Query instance.
* **Entity:** The entities are the persistence objects stores as a record in the database.
* **Persistence Unit:** It defines a set of all entity classes. In an application, EntityManager instances manage it. The set of entity classes represents the data contained within a single data store.
* **EntityTransaction:** It has a **one-to-one** relationship with the EntityManager class. For each EntityManager, operations are maintained by EntityTransaction class.
* **Query:** It is an interface that is implemented by each JPA vendor to obtain relation objects that meet the criteria.



JPA Class Relationships

The classes and interfaces that we have discussed above maintain a relationship. The following figure shows the relationship between classes and interfaces.



* The relationship between EntityManager and EntiyTransaction is **one-to-one**. There is an EntityTransaction instance for each EntityManager operation.
* The relationship between EntityManageFactory and EntiyManager is **one-to-many**. It is a factory class to EntityManager instance.
* The relationship between EntityManager and Query is **one-to-many**. We can execute any number of queries by using an instance of EntityManager class.
* The relationship between EntityManager and Entity is **one-to-many**. An EntityManager instance can manage multiple Entities.

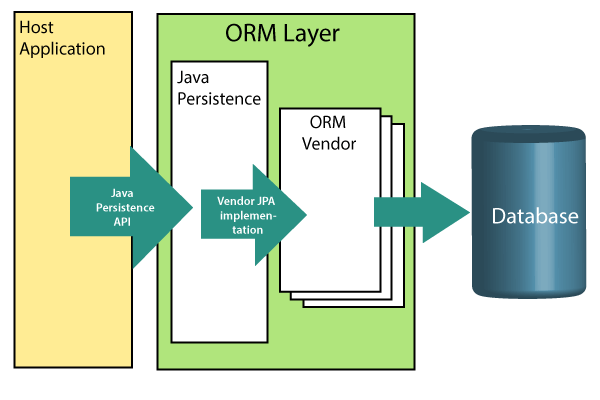
JPA Implementations

JPA is an open-source API. There is various enterprises vendor such as Eclipse, RedHat, Oracle, etc. that provides new products by adding the JPA in them. There are some popular JPA implementations frameworks such as **Hibernate, EclipseLink, DataNucleus,** etc. It is also known as **Object-Relation Mapping** (ORM) tool.

Object-Relation Mapping (ORM)

In ORM, the mapping of Java objects to database tables, and vice-versa is called **Object-Relational Mapping.** The ORM mapping works as a bridge between a **relational database** (tables and records) and **Java application** (classes and objects).

In the following figure, the ORM layer is an adapter layer. It adapts the language of object graphs to the language of SQL and relation tables.



The ORM layer exists between the application and the database. It converts the Java classes and objects so that they can be stored and managed in a relational database. By default, the name that persists become the name of the table, and fields become columns. Once an application sets-up, each table row corresponds to an object.

JPA Versions

Earlier versions of EJB defines the persistence layer combined with the business logic layer using **javax.ejb.EntityBean** Interface. EJB specification includes the definition of JPA.

While introducing EJB 3.0, the persistence layer was separated and specified as JPA 1.0 (Java Persistence API). The specifications of this API were released along with the specifications of JAVA EE5 on May 11, 2006, using JSR 220.

In 2019, JPA renamed to **Jakarta Persistence**. The latest version of JPA is **2.2**. It supports the following features:

* Java 8, data and time API
* CDI Injection in AttributeConvertes
* It makes annotations @Repeatable

Difference between JPA and Hibernate

**JPA:** JPA is a Java specification that is used to access, manage, and persist data between Java object and relational database. It is a standard approach for ORM.

**Hibernate:** It is a lightweight, open-source ORM tool that is used to store Java objects in the relational database system. It is a provider of JPA. It follows a common approach provided by JPA.

The following table describes the differences between JPA and Hibernate.

|  |  |
| --- | --- |
| **JPA** | **Hibernate** |
| JPA is a **Java specification** for mapping relation data in Java application. | Hibernate is an **ORM framework** that deals with data persistence. |
| JPA does not provide any implementation classes. | It provides implementation classes. |
| It uses platform-independent query language called **JPQL** (Java Persistence Query Language). | It uses its own query language called **HQL** (Hibernate Query Language). |
| It is defined in **javax.persistence** package. | It is defined in **org.hibernate** package. |
| It is implemented in various ORM tools like **Hibernate, EclipseLink,** etc. | Hibernate is the **provider** of JPA. |
| JPA uses **EntityManager** for handling the persistence of data. | In Hibernate uses **Session** for handling the persistence of data. |

Spring Boot Starter Data JPA

Spring Boot provides starter dependency **spring-boot-starter-data-jpa** to connect Spring Boot application with relational database efficiently. The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-data-jpa**</artifactId>**
4. **<version>**2.2.2.RELEASE**</version>**
5. **</dependency>**

Spring Boot JPA Example

Let's create a Spring Boot application that uses JPA to connect to the database. In the following example, we have used in-memory database **Apache Derby.**

**Apache Derby:** It is an **open-source, embedded** relational database implemented entirely in Java. It is available under the Apache License 2.0. There are following advantages of Apache Derby:

* It is easy to install, deploy, and use.
* It is based on Java, JDBC, and SQL standards.
* It provides an embedded JDBC driver that allows us to embed Derby in any Java-based solution.
* It also supports client/server mode with the Derby Network Client JDBC driver, and Derby Network Server.

Spring Boot can auto-configure an embedded database such as **H2, HSQL,**and**Derbydatabases**. We do not need to provide any connection URLs. We need only include a build dependency on the embedded database that we want to use.

In Spring Boot, we can easily integrate Apache Derby database just by adding **Derby** dependency in pom.xml file.

1. **<dependency>**
2. **<groupId>**org.apache.derby**</groupId>**
3. **<artifactId>**derby**</artifactId>**
4. **<scope>**runtime**</scope>**
5. **</dependency>**

**Step 1:** Open Spring Initializr <https://start.spring.io/>.

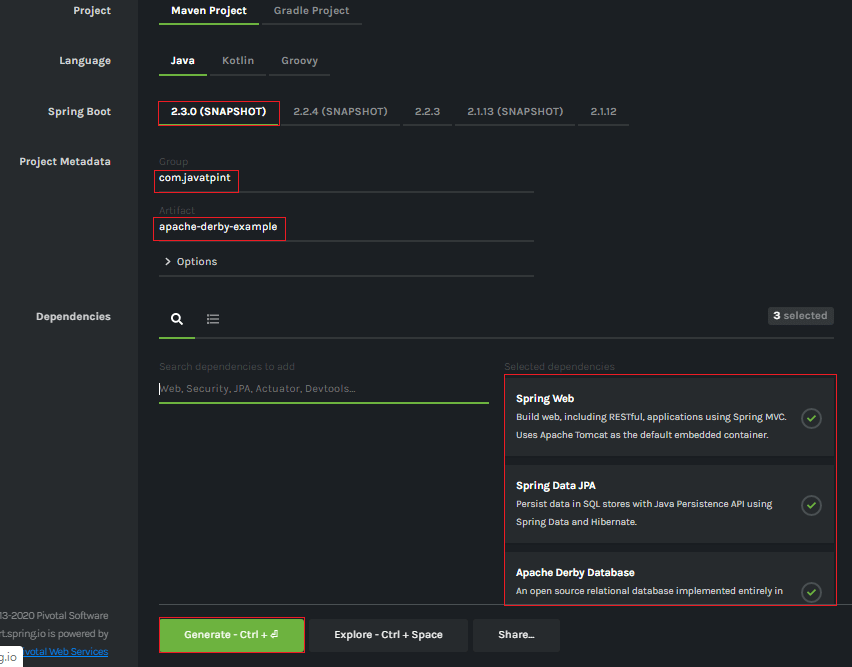
**Step 2:** Select the latest version of Spring Boot **2.3.0(SNAPSHOT)**

**Step 3:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 4:** Provide the **Artifact** Id. We have provided **apache-derby-example**.

**Step 5:** Add the dependencies: **Spring Web, Spring Data JPA,** and **Apache Derby Database**.

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the project in a Jar file and downloads it to the local system.



**Step 7: Extract** the Jar file and paste it into the STS workspace.

**Step 8: Import** the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder apache-derby-example -> Finish

It takes some time to import.

**Step 9:** Create a package with the name **com.javatpoint.model** in the folder **src/main/java.**

**Step 10:** Create a class with the name **UserRecord** in the package **com.javatpoint.model** and do the following:

* Define three variables **id, name,**and **email**.
* Generate Getters and Setter.  
  Right-click on the file -> Source -> Generate Getters and Setters
* Define a default constructor.
* Mark the class as an **Entity** by using the annotation **@Entity.**
* Mark **Id** as the primary key by using the annotation**@Id.**

**UserRecord.java**

1. **package** com.javatpoint.model;
2. **import** javax.persistence.Entity;
3. **import** javax.persistence.Id;
4. @Entity
5. **public** **class** UserRecord
6. {
7. @Id
8. **private** **int** id;
9. **private** String name;
10. **private** String email;
11. //default conatructor
12. **public** UserRecord()
13. {
14. }
15. **public** **int** getId()
16. {
17. **return** id;
18. }
19. **public** **void** setId(**int** id)
20. {
21. **this**.id = id;
22. }
23. **public** String getName()
24. {
25. **return** name;
26. }
27. **public** **void** setName(String name)
28. {
29. **this**.name = name;
30. }
31. **public** String getEmail()
32. {
33. **return** email;
34. }
35. **public** **void** setEmail(String email)
36. {
37. **this**.email = email;
38. }
39. }

**Step 11:** Create a package with the name **com.javatpoint.controller** in the folder **src/main/java.**

**Step 12:** Create a Controller class with the name **UserController** in the package **com.javatpoint.controller** and do the following:

* Mark the class as a controller by using the annotation **@RestController.**
* Autowired the class **UserService** by using the annotation **@Autowired**.
* We have defined two mappings, one for **getting all users** and the other for **add-user.**

**UserController.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.web.bind.annotation.RequestBody;
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. **import** com.javatpoint.model.UserRecord;
8. **import** com.javatpoint.service.UserService;
9. **import** java.util.List;
10. @RestController
11. **public** **class** UserController
12. {
13. @Autowired
14. **private** UserService userService;
15. @RequestMapping("/")
16. **public** List<UserRecord> getAllUser()
17. {
18. **return** userService.getAllUsers();
19. }
20. @RequestMapping(value="/add-user", method=RequestMethod.POST)
21. **public** **void** addUser(@RequestBody UserRecord userRecord)
22. {
23. userService.addUser(userRecord);
24. }
25. }

**Step 13:** Create a package with the name **com.javatpoint.service** in the folder **src/main/java.**

**Step 14:** Create a Service class with the name **UserService** in the package **com.javatpoint.service** and do the following:

* Mark the class as service by using the annotation **@Service.**
* Autowired the **UserRepository**
* Define a method **getAllUsers()** that returns a List of
* Define another method name **addUser()** that saves the user record.

**UserService.java**

1. **package** com.javatpoint.service;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.stereotype.Service;
6. **import** com.javatpoint.model.UserRecord;
7. **import** com.javatpoint.repository.UserRepository;
8. @Service
9. **public** **class** UserService
10. {
11. @Autowired
12. **private** UserRepository userRepository;
13. **public** List<UserRecord> getAllUsers()
14. {
15. List<UserRecord>userRecords = **new** ArrayList<>();
16. userRepository.findAll().forEach(userRecords::add);
17. **return** userRecords;
18. }
19. **public** **void** addUser(UserRecord userRecord)
20. {
21. userRepository.save(userRecord);
22. }
23. }

**Step 15:** Create a package with the name **com.javatpoint.repository** in the folder **src/main/java.**

**Step 16:** Create a repository interface with the name **UserRepository**in the package **com.javatpoint.repository**and extends**CrudRepository**.

**UserRepository.java**

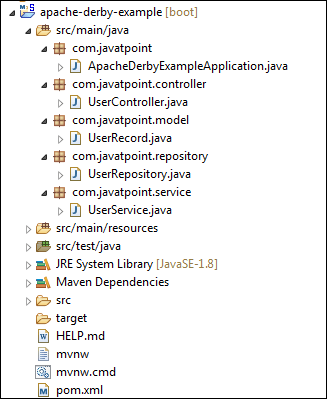
1. **package** com.javatpoint.repository;
2. **import** org.springframework.data.repository.CrudRepository;
3. **import** com.javatpoint.model.UserRecord;
4. **public** **interface** UserRepository **extends** CrudRepository<UserRecord, String>
5. {
6. }

**Step 17:** Now, open the **ApacheDerbyExampleApplication.java** file. It created by default when we set-up an application.

**ApacheDerbyExampleApplication.java**

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

Now, we have set-up all the necessary classes and packages according to the requirements. Notice that we have not provided any **connection URL** for the database. After completing all the above steps, the project directory looks like the following:



Let's run the application.

**Step 18:** Open the **ApacheDerbyExampleApplication.java** file and run it as Java Application.

**Step 19:** Open the browser and invoke the URL http://localhost:8080/. It returns an empty list because we have not added any user in the List.

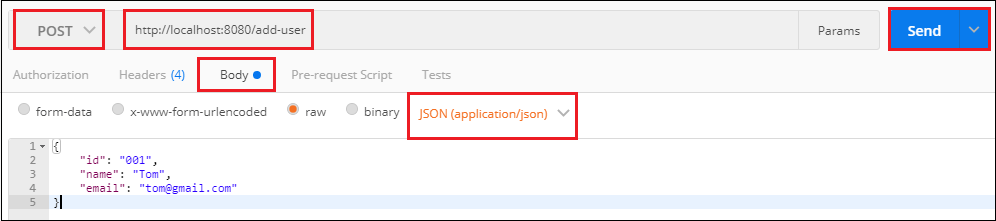
To add a user to the database, we will send a **POST** request by using the **Postman**.

**Step 20:** Open the **Postman** and do the following:

* Select the **POST**
* Invoke the URL http://localhost:8080/add-user.
* Click on the **Body**
* Select Content-Type as **JSON**(application/json).
* Insert the data which want to insert in the database. We have inserted the following data:

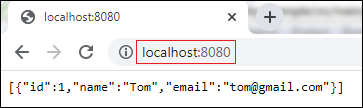
1. {
2. "id": "001",
3. "name": "Tom",
4. "email": "tom@gmail.com"
5. }

* Click on the **Send** button.



When we click on the Send button, it shows **Status:200 OK**. It means the request has been successfully executed.

**Step 21:** Open the browser and invoke the URL http://localhost:8080. It returns the user that we have inserted in the database.



# Spring Boot JDBC

**Spring Boot JDBC** provides starter and libraries for connecting an application with JDBC.

In Spring Boot JDBC, the database related beans such as **DataSource, JdbcTemplate,** and **NamedParameterJdbcTemplate** auto-configures and created during the startup. We can autowire these classes if we want to use it. For example:

1. @Autowired
2. JdbcTemplate jdbcTemplate;
3. @Autowired
4. **private** NamedParameterJdbcTemplate jdbcTemplate;

In **application.properties** file, we configure **DataSource** and **connection pooling**. [Spring Boot](https://www.javatpoint.com/spring-boot-tutorial)

chooses **tomcat** pooling by default.

## JDBC Connection Pooling

**JDBC connection pooling** is a mechanism that manages **multiple** database connection requests. In other words, it facilitates connection reuse, a memory cache of database connections, called a **connection pool.** A connection pooling module maintains it as a layer on top of any standard JDBC driver product.

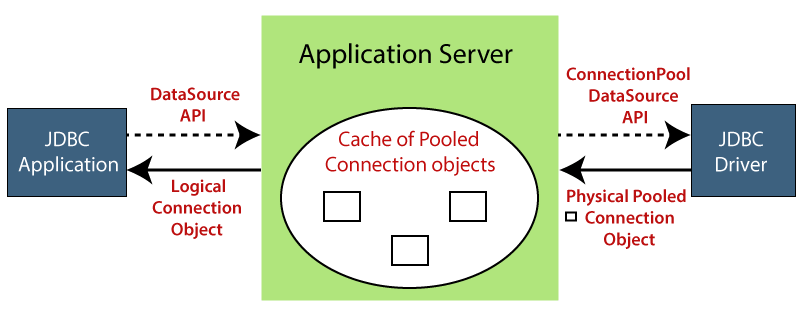
21.9M

414

Prime Ministers of India | List of Prime Minister of India (1947-2020)

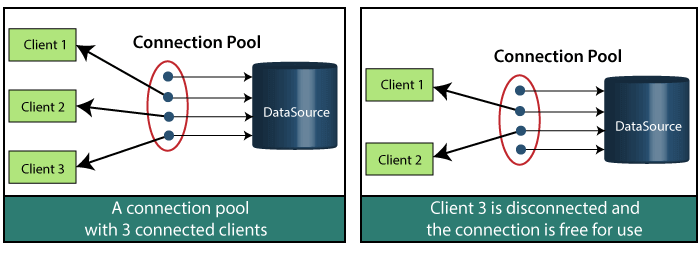
**Next**

**Stay**



It increases the speed of data access and reduces the number of database connections for an application. It also improves the performance of an application. Connection pool performs the following tasks:

* Manage available connection
* Allocate new connection
* Close connection



In the above figure, there are **clients, a connection pool** (that has four available connections), and **a** **DataSource**.

In the first figure, there are three clients connected with different connections, and a connection is available. In the second figure, Client 3 has disconnected, and that connection is available.

When a client completes his work, it releases the connection, and that connection is available for other clients.

### **HikariCP**

The default connection pool in Spring Boot 2 is **HikariCP**. It provides enterprise-ready features and better performance. HikariCP is a JDBC DataSource implementation that provides a connection pooling mechanism.

* If the HikariCP is present on the classpath, the Spring Boot automatically configures it.
* If the HikariCP is not found on the classpath, Spring Boot looks for the **Tomcat JDBC Connection Pool.**If it is on the classpath Spring Boot, pick it up.
* If both the above options are not available, Spring Boot chooses **Apache Commons DBCP2** as the JDBC connection pool.

We can also configure a connection pool manually, if we do not want to use the default connection pool. Suppose, we want to use Tomcat JDBC connection pool instead of HikariCP. We will exclude **HikariCP** dependency and add the **tomcat-jdbc** dependency in the pom.xml file, as shown below.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-data-jpa**</** **artifactId** **>**
4. **<exclusions>**
5. **<exclusion>**
6. **<groupId>**com.zaxxer**</groupId>**
7. **<artifactId>**HikariCP**</** **artifactId** **>**
8. **</exclusion>**
9. **</exclusions>**
10. **</dependency>**
11. **<dependency>**
12. **<groupId>**org.apache.tomcat**</groupId>**
13. **<artifactId>**tomcat-jdbc**</artifactId>**
14. **<version>**9.0.10**</version>**
15. **</dependency>**
16. **<dependency>**
17. **<groupId>**com.h2database**</groupId>**
18. **<artifactId>**h2**</artifactId>**
19. **<version>**1.4.9**</version>**
20. **<socpe>**runtime**</scoope>**
21. **</dependency>**

The above approach allows us to use Tomcat connection pool without having to write a **@Configuration** class and programmatically define a **DataSource** bean.

On the other hand, we can also skip the connection pool scanning algorithm that Spring Boot uses. We can explicitly specify a connection pooling datasource by adding the property **spring.datasource.type**in the application.properties file.

1. Spring.datasource.type=org.apache.tomcat.jdbc.pool.DataSource

We have set up the Tomcat connection pool. Now, we will add some properties in **application.properties**that optimize its performance and suiting some specific requirements.

1. spring.datasource.tomcat.initial-size=20
2. spring.datasource.tomcat.max-wait=25000
3. spring.datasource.tomcat.max-active=70
4. spring.datasource.tomcat.max-idle=20
5. spring.datasource.tomcat.min-idle=9
6. spring.datasource.tomcat.default-auto-commit=true

If we want to connect to [MySQL](https://www.javatpoint.com/mysql-tutorial)

database, we need to include the JDBC driver in the application's classpath:

1. <!-- MySQL JDBC driver -->
2. **<dependency>**
3. **<groupId>**mysql**</groupId>**
4. **<artifactId>**mysql-connector-java**</artifactId>**
5. **</dependency>**

After that, define the **datasoure** properties in **application.properties**file.

Use the following properties if you are using **MySQL** database:

1. spring.datasource.url=jdbc:mysql://192.168.1.4:3306/test
2. spring.datasource.username=javatpoint
3. spring.datasource.password=password

Use the following properties if you are using **Oracle** database:

1. spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
2. spring.datasource.username=system
3. spring.datasource.password=Password123

#### **Note: Spring Boot 2 uses HikariCP as the database connection pool, by default. If the HikariCP is not present in the classpath, Spring Boot choose tomcat pooling by default.**

## Why should we use Spring Boot JDBC?

The functionality of [Spring JDBC](https://www.javatpoint.com/spring-tutorial)

and Spring Boot JDBC is the same except the implementations. There are following advantages of Spring Boot JBDC over Spring JDBC:

|  |  |
| --- | --- |
| **Spring Boot JDBC** | **Spring JDBC** |
| There is only a **spring-boot-starter-jdbc** dependency is required. | In Spring JDBC, multiple dependencies need to be configured like **spring-jdbc** and **spring-context.** |
| It automatically configures Datasource bean, if not maintain explicitly. If we do not want to use the bean, we can set a property **spring.datasource.initialize** to **false**. | In Spring JDBC, it is necessary to create a database bean either using **XML** or **javaconfig**. |
| We do not need to register Template beans because Spring Boot automatically registers beans. | The Template beans such as **PlatformTransactionManager, JDBCTemplate, NamedParameterJdbcTemplate** must be registered. |
| Any db initialization scripts stored in .sql file gets executed automatically. | If any db initialization scripts like dropping or creation of tables are created in SQL file, this info needs to be given explicitly in the configuration. |

## JDBC vs. Hibernate

|  |  |
| --- | --- |
| **JDBC** | **Hibernate** |
| JDBC is a **technology**. | Hibernate is an **ORM** framework. |
| In JDBC, the user is responsible for creating and closing the connections. | In Hibernate, the run time system takes care of creating and closing the connections. |
| It does not support lazy loading. | It supports lazy loading that offers better performance. |
| It does not support associations (the connection between two separate classes). | It supports associations. |

In the next section, we will learn the connectivity of MySQL in an Spring Boot application.

# Spring Boot JDBC Example

Spring Boot provides starter and libraries for connecting to our application with JDBC. Here, we are creating an application which connects with Mysql database. It includes the following steps to create and setup [JDBC with Spring Boot](https://www.javatpoint.com/spring-boot-jdbc)

.

Create a database

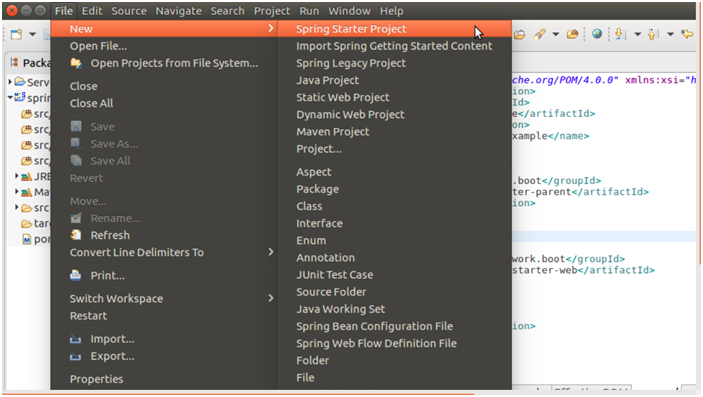
1. create database springbootdb

Create a table in to [mysql](https://www.javatpoint.com/mysql-tutorial)

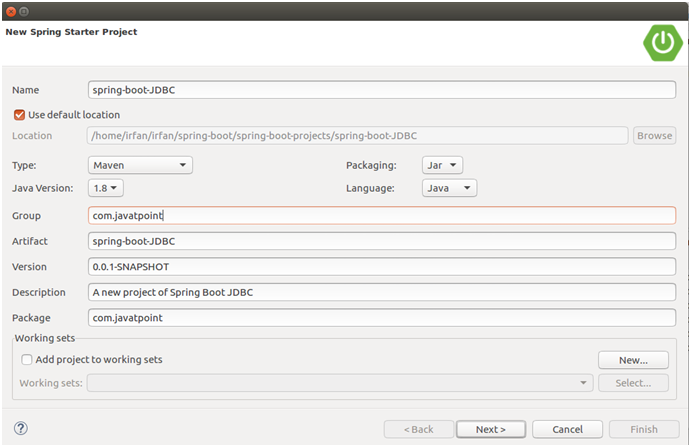
1. create table user(id **int** UNSIGNED primary key not **null** auto\_increment, name varchar(100), email varchar(100));

Creating a [Spring Boot](https://www.javatpoint.com/spring-boot-tutorial)

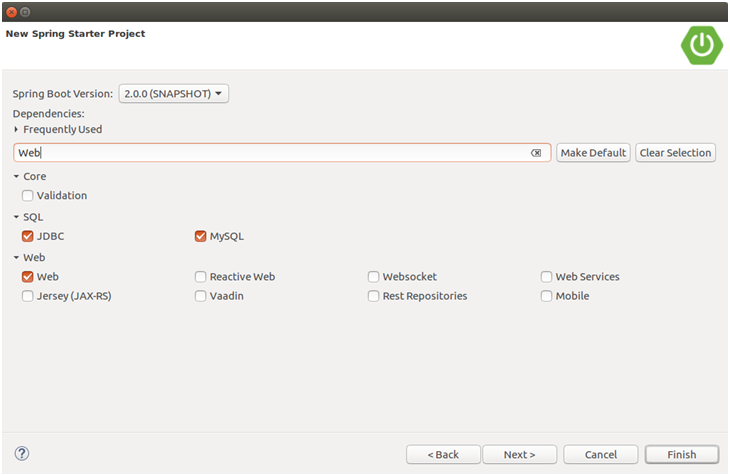
Pproject



Providing project name and other project related information.



Providing dependencies



After finishing, create following files in your project.

Configure database into application.properties file.

// application.properties

1. spring.datasource.url=jdbc:mysql://localhost:3306/springbootdb
2. spring.datasource.username=root
3. spring.datasource.password=mysql
4. spring.jpa.hibernate.ddl-auto=create-drop

// SpringBootJdbcApplication.java

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootJdbcApplication {
6. **public** **static** **void** main(String[] args) {
7. SpringApplication.run(SpringBootJdbcApplication.**class**, args);
8. }
9. }

Creating a controller to handle HTTP requests.

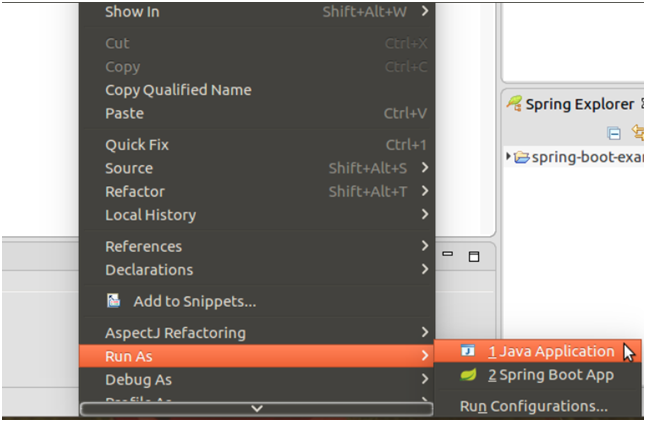
// SpringBootJdbcController.java

1. **package** com.javatpoint;
2. **import** org.springframework.web.bind.annotation.RequestMapping;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.jdbc.core.JdbcTemplate;
5. **import** org.springframework.web.bind.annotation.RestController;
6. @RestController
7. **public** **class** SpringBootJdbcController {
8. @Autowired
9. JdbcTemplate jdbc;
10. @RequestMapping("/insert")
11. **public** String index(){
12. jdbc.execute("insert into user(name,email)values('javatpoint','java@javatpoint.com')");
13. **return**"data inserted Successfully";
14. }
15. }

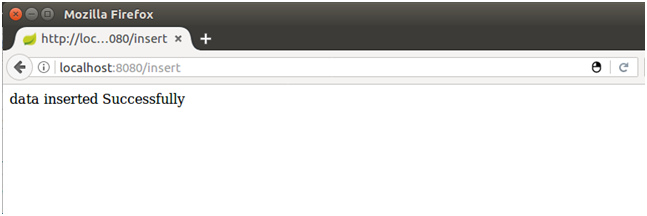
### **Run the application**

Run **SpringBootJdbcApplication.java** file as [Java](https://www.javatpoint.com/java-tutorial)

application.



Now, open browser and follow the following URL.



It says data has inserted successfully. let's confirm it by checking mysql table.



Well, our application is working fine. Now, we can perform other database operations as well.

# Spring Boot H2 Database

## What is the in-memory database

In-memory database relies on system memory as oppose to disk space for storage of data. Because memory access is faster than disk access. We use the in-memory database when we do not need to persist the data. The in-memory database is an embedded database. The in-memory databases are volatile, by default, and all stored data loss when we restart the application.

The widely used in-memory databases are **H2, HSQLDB**(HyperSQL Database)**,**and**Apache Derby.**It creates the configuration automatically.

## Persistence vs. In-memory Database

The persistent database persists the data in physical memory. The data will be available even if the database server is bounced. Some popular persistence databases are [**Oracle**](https://www.javatpoint.com/oracle-tutorial)**,**[**MySQL**](https://www.javatpoint.com/mysql-tutorial)**,**[**Postgres**](https://www.javatpoint.com/postgresql-tutorial)**,** etc.

In the case of the **in-memory database,** data store in the **system memory**. It lost the data when the program is closed. It is helpful for **POC**s (Proof of Concepts), not for a production application. The widely used in-memory database is **H2.**

## What is the H2 Database

**H2** is an **embedded, open-source,**and**in-memory** database. It is a relational database management system written in [Java](https://www.javatpoint.com/java-tutorial). It is a **client/server** application. It is generally used in **unit testing**. It stores data in memory, not persist the data on disk.

**Advantages**

* Zero configuration
* It is easy to use.
* It is lightweight and fast.
* It provides simple Configuration to switch between a real database and in-memory database.
* It supports standard SQL and JDBC API.
* It provides a web console to maintain in the database.

## Configure H2 Database

If we want to use H2 database in an application we need to add the following dependency in pom.xml file:

1. **<dependency>**
2. **<groupId>**com.h2database**</groupId>**
3. **<artifactId>**h2**</artifactId>**
4. **<scope>**runtime**</scope>**
5. **</dependency>**

After adding the dependency, we need to configure **data source URL, driver class name, username,** and **password** of H2 database. Spring Boot provide an easy way to configure these properties in **application.properties** file.

1. spring.datasource.url=jdbc:h2:mem:testdb
2. spring.datasource.driverClassName=org.h2.Driver
3. spring.datasource.username=sa
4. spring.datasource.**password**=
5. spring.jpa.**database**-platform=org.hibernate.dialect.H2Dialect

In the **spring.datasource.url** property, **mem** is the name of an in-memory database and **testdb** is the name of schema that H2 provides, by default. We can also define our own schema and database. The default username is **sa** and the blank password denotes an **empty** password. If we want to change the username and password, we can override these values.

## Persist the data in H2 Database

If we want to persist the data in the H2 database, we should store data in a file. To achieve the same, we need to change the datasource [URL](https://www.javatpoint.com/url-full-form) property.

1. #persist the data
2. spring.datasource.url=jdbc:h2:file:/data/sampledata
3. spring.datasource.url=jdbc:h2:C:/data/sampledata

In the above property, the **sampledata** is a file name.

## Create Schema and Populate Data

We can define schema by creating a [**SQL**](https://www.javatpoint.com/sql-tutorial) file in the **resource** folder (src/main/resource).

**schema.sql**

1. **DROP** **TABLE** IF EXISTS CITY;
2. **CREATE** **TABLE** CITY (
3. City\_code **INT** AUTO\_INCREMENT  **PRIMARY** **KEY**,
4. city\_name **VARCHAR**(50) NOT NULL,
5. city\_pincode **INT**(8) NOT NULL
6. );

We can populate data in the table by creating a **SQL** file in the **resource** folder (src/main/resource).

**data.sql**

1. **INSERT** **INTO** CITY **VALUES** (11, 'Delhi', 110001);
2. **INSERT** **INTO** CITY **VALUES** (12, 'Kanpur', 208001);
3. **INSERT** **INTO** CITY **VALUES** (13, 'Lucknow', 226001);

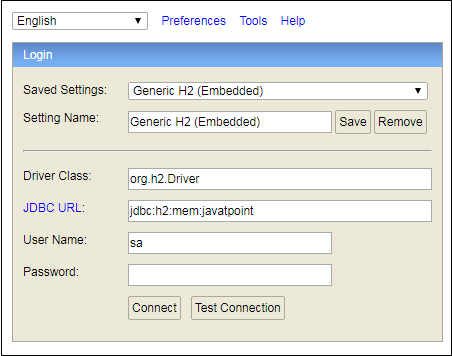
Spring Boot automatically picks up the **data.sql** file and run it against the H2 database during the application startup.

## H2 Console

By default, the console view of the H2 database is disabled. Before accessing the H2 database, we must enable it by using the following property.

1. #enabling the H2 console
2. spring.h2.console.enabled=**true**

Once we have enabled the H2 console, now we can access the H2 console in the browser by invoking the URL http://localhost:8080/h2-console. The following figure shows the console view of the H2 database.



In the above screenshot, we have defined our own database named **javatpoint**.

## Spring Boot H2 Example

Let's set up a Spring Boot application with the H2 database.

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

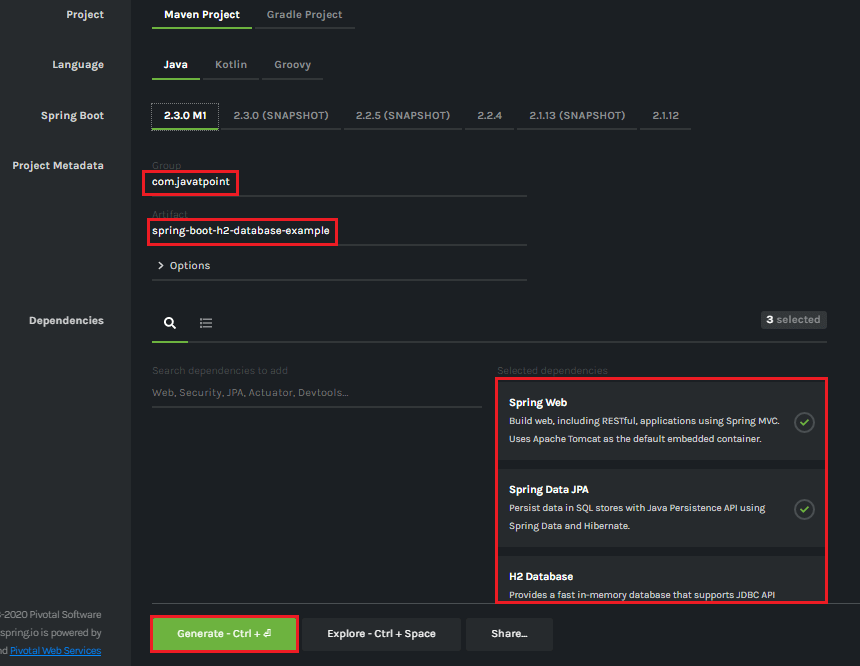
**Step 2:** Select the Spring Boot version **2.3.0.M1.**

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-h2-database-example.**

**Step 5:** Add the dependencies **Spring Web, Spring Data**[**JPA**](https://www.javatpoint.com/jpa-tutorial)**,**and**H2 Database.**

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the project in a **Jar** file and downloads it to the local system.



**Step 7:** **Extract** the Jar file and paste it into the STS workspace.

**Step 8:** **Import** the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-h2-database-example -> Finish

It takes some time to import.

**Step 9:** Create a package with the name **com.javatpoint.model**in the folder **src/main/java.**

**Step 10:** Create a model class in the package **com.javatpoint.model.**We have created model class with the name **Student.**In the Books class, we have done the following:

* Define four variable **id, age, name,**and
* Generate Getters and Setters.  
  Right-click on the file -> Source -> Generate Getters and Setters.
* Mark the class as **Entity** by using the annotation **@Entity.**
* Mark the class as **Table** name by using the annotation **@Table.**
* Define each variable as **Column** by using the annotation **@Column.**

**Student.java**

1. **package** com.javatpoint.model;
2. **import** javax.persistence.Column;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.Id;
5. **import** javax.persistence.Table;
6. //mark class as an Entity
7. @Entity
8. //defining class name as Table name
9. @Table
10. **public** **class** Student
11. {
12. //mark id as primary key
13. @Id
14. //defining id as column name
15. @Column
16. **private** **int** id;
17. //defining name as column name
18. @Column
19. **private** String name;
20. //defining age as column name
21. @Column
22. **private** **int** age;
23. //defining email as column name
24. @Column
25. **private** String email;
26. **public** **int** getId()
27. {
28. **return** id;
29. }
30. **public** **void** setId(**int** id)
31. {
32. **this**.id = id;
33. }
34. **public** String getName()
35. {
36. **return** name;
37. }
38. **public** **void** setName(String name)
39. {
40. **this**.name = name;
41. }
42. **public** **int** getAge()
43. {
44. **return** age;
45. }
46. **public** **void** setAge(**int** age)
47. {
48. **this**.age = age;
49. }
50. **public** String getEmail()
51. {
52. **return** email;
53. }
54. **public** **void** setEmail(String email)
55. {
56. **this**.email = email;
57. }
58. }

**Step 11:** Create a package with the name **com.javatpoint.controller**in the folder **src/main/java.**

**Step 12:** Create a Controller class in the package **com.javatpoint.controller**. We have created controller class with the name **StudentController**. In the StudentController class, we have done the following:

* Mark the class as **RestController**by using the annotation**@RestController.**
* Autowire the **StudentService** class by using the annotation **@Autowired**.
* Define the following methods:
  + **getAllStudent():** It returns a List of all Students.
  + **getStudent():** It returns a student detail that we have specified in the path variable. We have passed id as an argument by using the annotation @PathVariable. The annotation indicates that a method parameter should be bound to a URI template variable.
  + **deleteStudent():** It deletes a specific student that we have specified in the path variable.
  + **saveStudent():** It saves the student detail. The annotation @RequestBody indicates that a method parameter should be bound to the body of the web request.

**StudentController.java**

1. **package** com.javatpoint.controller;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.DeleteMapping;
5. **import** org.springframework.web.bind.annotation.GetMapping;
6. **import** org.springframework.web.bind.annotation.PathVariable;
7. **import** org.springframework.web.bind.annotation.PostMapping;
8. **import** org.springframework.web.bind.annotation.RequestBody;
9. **import** org.springframework.web.bind.annotation.RestController;
10. **import** com.javatpoint.model.Student;
11. **import** com.javatpoint.service.StudentService;
12. //creating RestController
13. @RestController
14. **public** **class** StudentController
15. {
16. //autowired the StudentService class
17. @Autowired
18. StudentService studentService;
19. //creating a get mapping that retrieves all the students detail from the database
20. @GetMapping("/student")
21. **private** List<Student> getAllStudent()
22. {
23. **return** studentService.getAllStudent();
24. }
25. //creating a get mapping that retrieves the detail of a specific student
26. @GetMapping("/student/{id}")
27. **private** Student getStudent(@PathVariable("id") **int** id)
28. {
29. **return** studentService.getStudentById(id);
30. }
31. //creating a delete mapping that deletes a specific student
32. @DeleteMapping("/student/{id}")
33. **private** **void** deleteStudent(@PathVariable("id") **int** id)
34. {
35. studentService.delete(id);
36. }
37. //creating post mapping that post the student detail in the database
38. @PostMapping("/student")
39. **private** **int** saveStudent(@RequestBody Student student)
40. {
41. studentService.saveOrUpdate(student);
42. **return** student.getId();
43. }
44. }

**Step 13:** Create a package with the name **com.javatpoint.service**in the folder **src/main/java.**

**Step 14:** Create a **Service** class. We have created a service class with the name **StudentService** in the package **com.javatpoint.service.**

**StudentService.java**

1. **package** com.javatpoint.service;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.stereotype.Service;
6. **import** com.javatpoint.model.Student;
7. **import** com.javatpoint.repository.StudentRepository;
8. @Service
9. **public** **class** StudentService
10. {
11. @Autowired
12. StudentRepository studentRepository;
13. //getting all student records
14. **public** List<Student> getAllStudent()
15. {
16. List<Student> students = **new** ArrayList<Student>();
17. studentRepository.findAll().forEach(student -> students.add(student));
18. **return** students;
19. }
20. //getting a specific record
21. **public** Student getStudentById(**int** id)
22. {
23. **return** studentRepository.findById(id).get();
24. }
25. **public** **void** saveOrUpdate(Student student)
26. {
27. studentRepository.save(student);
28. }
29. //deleting a specific record
30. **public** **void** delete(**int** id)
31. {
32. studentRepository.deleteById(id);
33. }
34. }

**Step 15:** Create a package with the name **com.javatpoint.repository**in the folder **src/main/java.**

**Step 16:** Create a **Repository** interface. We have created a repository interface with the name **StudentRepository**in the package **com.javatpoint.repository.**It extends the **Crud Repository** interface.

**StudentRepository.java**

1. **package** com.javatpoint.repository;
2. **import** org.springframework.data.repository.CrudRepository;
3. **import** com.javatpoint.model.Student;
4. **public** **interface** StudentRepository **extends** CrudRepository<Student, Integer>
5. {
6. }

Now we will configure the datasource **URL, driver class name, username,**and**password,**in the **application.properties**file.

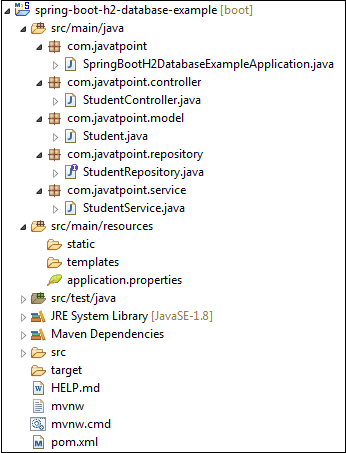
**Step 17:** Open the **application.properties** file and configure the following properties.

**application.properties**

1. spring.datasource.url=jdbc:h2:mem:javatpoint
2. spring.datasource.driverClassName=org.h2.Driver
3. spring.datasource.username=sa
4. spring.datasource.**password**=
5. spring.jpa.**database**-platform=org.hibernate.dialect.H2Dialect
6. #enabling the H2 console
7. spring.h2.console.enabled=**true**

#### **Note: Do not forget to enable the H2 console.**

After creating all the classes and packages, the project directory looks like the following.



Now we will run the application.

**Step 18:** Open **SpringBootH2DatabaseExampleApplication.java**file and run it as Java Application.

**SpringBootH2DatabaseExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootH2DatabaseExampleApplication
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(SpringBootH2DatabaseExampleApplication.**class**, args);
10. }
11. }

In the next step, we will use rest client **Postman** for sending the **POST**and**GET** request**.**If the Postman is not installed in your system, follow the steps below:

* Download the Postman from <https://www.getpostman.com/downloads/> or add Google Chrome extension in the browser <https://bit.ly/1HCOCwF>.
* Launch the Postman and **Signup**. Create a user name. We have created user with the name **javatpoint**and clicked on **Submit**

**Step 19:** Open the **Postman**and do the following:

* Select the **POST**
* Invoke the URL http://localhost:8080/student.
* Select the **Body**
* Select he Content-Type **JSON (application/json).**
* Insert the data. We have inserted the following data in the body:

1. {
2. "id": "001",
3. "age": "23",
4. "name": "Amit",
5. "email": "amit@yahoo.co.in"
6. }

* Click on the **Send**

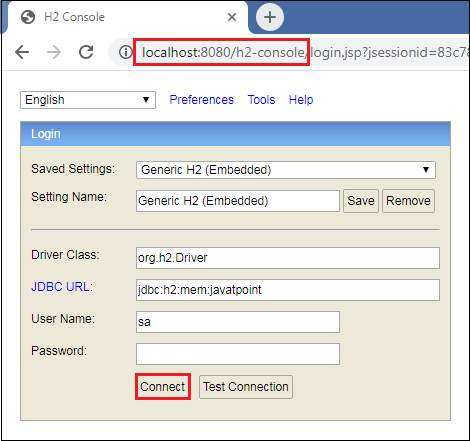
When the request is successfully executed, it shows the **Status:200 OK**. It means the record has been successfully inserted in the database.

Similarly, we have inserted the following data.

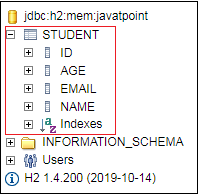
1. {
2. "id": "002",
3. "age": "24",
4. "name": "Vadik",
5. "email": "vadik@yahoo.co.in"
6. }
7. {
8. "id": "003",
9. "age": "21",
10. "name": "Prateek",
11. "email": "prateek@yahoo.co.in"
12. }
13. {
14. "id": "004",
15. "age": "25",
16. "name": "Harsh",
17. "email": "harsh@yahoo.co.in"
18. }
19. {
20. "id": "005",
21. "age": "24",
22. "name": "Swarit",
23. "email": "Swarit@yahoo.co.in"
24. }

Let's access the H2 console to see the data.

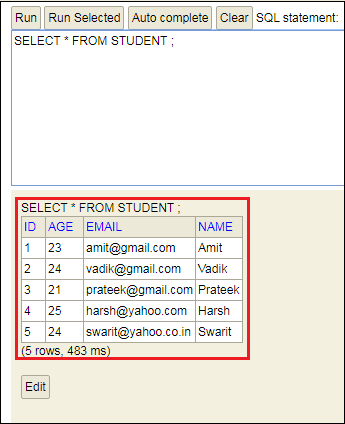
**Step 20:** Open the browser and invoke the URL http://localhost:8080/h2-console. Click on the **Connect** button, as shown below.



After clicking on the **Connect** button, we see the **Student** table in the database, as shown below.



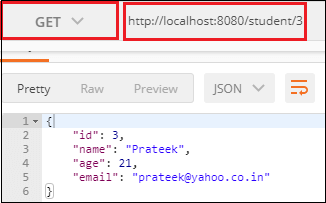
**Step 21:** Click on the **Student** table and then click on the **Run** button. The table shows the data that we have inserted in the body.



**Step 22:** Open the Postman and send a **GET** request. It returns the data that we have inserted in the database.

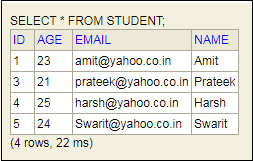


Let's send a **GET** request with the URL http://localhost:8080/student/{id}. We have invoked the URL http://localhost:8080/student/3. It returns the detail of the student whose id is 3.



Similarly, we can also send a **DELETE** request. Suppose we want to delete a student record whose id is 2.

To delete a student record, send a **DELETE** request with the URL http://localhost:8080/student/2. We see that the student whose id is **2** has been deleted from the database.



# Spring Boot CRUD Operations

## What is the CRUD operation?

The **CRUD** stands for **Create, Read/Retrieve, Update,** and **Delete**. These are the four basic functions of the persistence storage.

The CRUD operation can be defined as user interface conventions that allow view, search, and modify information through computer-based forms and reports. CRUD is data-oriented and the standardized use of **HTTP action verbs**. HTTP has a few important verbs.

* **POST:** Creates a new resource
* **GET:** Reads a resource
* **PUT:** Updates an existing resource
* **DELETE:** Deletes a resource

Within a database, each of these operations maps directly to a series of commands. However, their relationship with a RESTful API is slightly more complex.

## Standard CRUD Operation

* **CREATE Operation:** It performs the INSERT statement to create a new record.
* **READ Operation:** It reads table records based on the input parameter.
* **UPDATE Operation:** It executes an update statement on the table. It is based on the input parameter.
* **DELETE Operation:** It deletes a specified row in the table. It is also based on the input parameter.

## How CRUD Operations Works

CRUD operations are at the foundation of the most dynamic websites. Therefore, we should differentiate **CRUD** from the **HTTP** **action verbs**.

32.6M

564

Hello Java Program for Beginners

Suppose, if we want to **create** a new record, we should use HTTP action verb **POST**. To **update** a record, we should use the **PUT** verb. Similarly, if we want to **delete** a record, we should use the **DELETE**verb. Through CRUD operations, users and administrators have the right to retrieve, create, edit, and delete records online.

We have many options for executing CRUD operations. One of the most efficient choices is to create a set of stored procedures in SQL to execute operations.

The CRUD operations refer to all major functions that are implemented in relational database applications. Each letter of the CRUD can map to a SQL statement and HTTP methods.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **SQL** | **HTTP verbs** | **RESTful Web Service** |
| **Create** | INSERT | PUT/POST | POST |
| **Read** | SELECT | GET | GET |
| **Update** | UPDATE | PUT/POST/PATCH | PUT |
| **Delete** | DELETE | DELETE | DELETE |

## Spring Boot CrudRepository

Spring Boot provides an interface called **CrudRepository**that contains methods for CRUD operations. It is defined in the package **org.springframework.data.repository**. It extends the Spring Data**Repository** interface. It provides generic Crud operation on a repository. If we want to use CrudRepository in an application, we have to create an interface and extend the **CrudRepository**.

**Syntax**

1. **public** **interface** CrudRepository<T,ID> **extends** Repository<T,ID>

where,

* **T** is the domain type that repository manages.
* **ID** is the type of the id of the entity that repository manages.

For example:

1. **public** **interface** StudentRepository **extends** CrudRepository<Student, Integer>
2. {
3. }

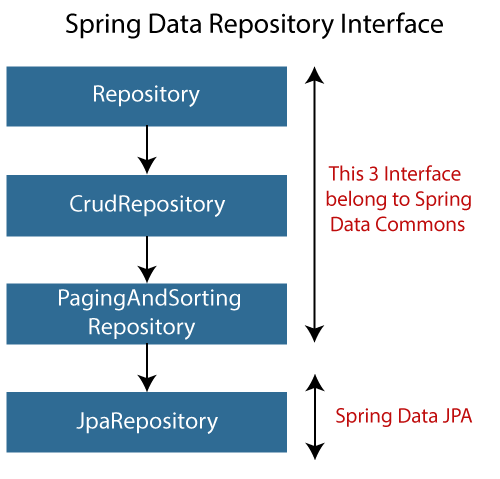
In the above example, we have created an interface named **StudentRepository** that extends CrudRepository. Where **Student** is the repository to manage, and **Integer** is the type of Id that is defined in the Student repository.

## Spring Boot JpaRepository

JpaRepository provides JPA related methods such as flushing, persistence context, and deletes a record in a batch. It is defined in the package **org.springframework.data.jpa.repository.**JpaRepository extends both **CrudRepository** and **PagingAndSortingRepository.**

For example:

1. **public** **interface** BookDAO **extends** JpaRepository
2. {
3. }



## Why should we use these interfaces?

* The interfaces allow Spring to find the repository interface and create proxy objects for that.
* It provides methods that allow us to perform some common operations. We can also define custom methods as well.

## CrudRepository vs. JpaRepository

|  |  |
| --- | --- |
| **CrudRepository** | **JpaRepository** |
| CrudRepository does not provide any method for pagination and sorting. | JpaRepository extends PagingAndSortingRepository. It provides all the methods for implementing the pagination. |
| It works as a **marker** interface. | JpaRepository extends both **CrudRepository** and **PagingAndSortingRepository**. |
| It provides CRUD function only. For example **findById(), findAll(),** etc. | It provides some extra methods along with the method of PagingAndSortingRepository and CrudRepository. For example, **flush(), deleteInBatch().** |
| It is used when we do not need the functions provided by JpaRepository and PagingAndSortingRepository. | It is used when we want to implement pagination and sorting functionality in an application. |

## Spring Boot CRUD Operation Example

Let's set up a Spring Boot application and perform CRUD operation.

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

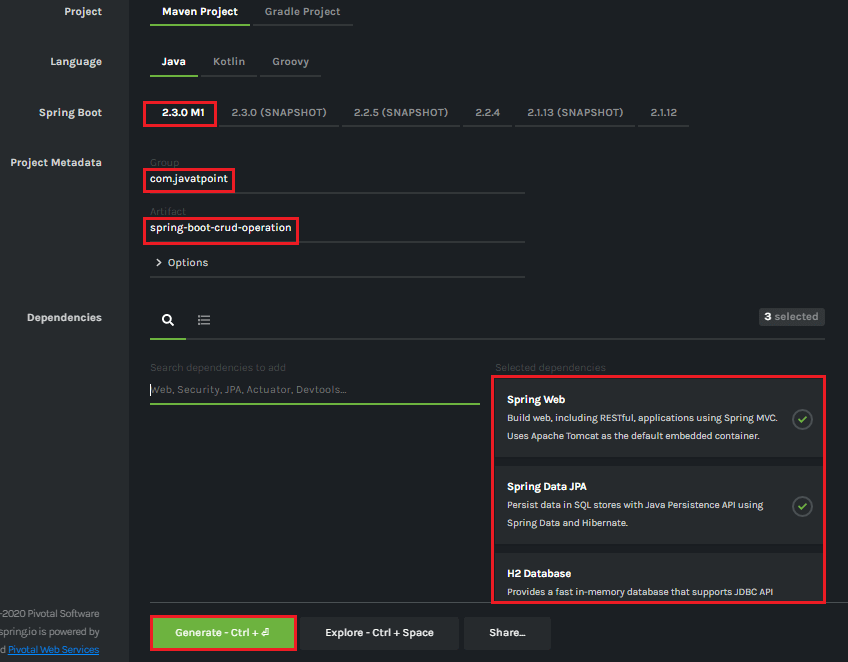
**Step 2:** Select the Spring Boot version **2.3.0.M1.**

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-crud-operation.**

**Step 5:** Add the dependencies **Spring Web, Spring Data JPA,**and**H2 Database.**

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the specifications in a **Jar** file and downloads it to the local system.



**Step 7: Extract** the Jar file and paste it into the STS workspace.

**Step 8: Import** the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-crud-operation -> Finish

It takes some time to import.

**Step 9:** Create a package with the name **com.javatpoint.model**in the folder **src/main/java.**

**Step 10:** Create a model class in the package **com.javatpoint.model.**We have created a model class with the name **Books.**In the Books class, we have done the following:

* Define four variable **bookid, bookname, author,**and
* Generate Getters and Setters.  
  Right-click on the file -> Source -> Generate Getters and Setters.
* Mark the class as an **Entity** by using the annotation **@Entity.**
* Mark the class as **Table** name by using the annotation **@Table.**
* Define each variable as **Column** by using the annotation **@Column.**

**Books.java**

1. **package** com.javatpoint.model;
2. **import** javax.persistence.Column;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.Id;
5. **import** javax.persistence.Table;
6. //mark class as an Entity
7. @Entity
8. //defining class name as Table name
9. @Table
10. **public** **class** Books
11. {
12. //Defining book id as primary key
13. @Id
14. @Column
15. **private** **int** bookid;
16. @Column
17. **private** String bookname;
18. @Column
19. **private** String author;
20. @Column
21. **private** **int** price;
22. **public** **int** getBookid()
23. {
24. **return** bookid;
25. }
26. **public** **void** setBookid(**int** bookid)
27. {
28. **this**.bookid = bookid;
29. }
30. **public** String getBookname()
31. {
32. **return** bookname;
33. }
34. **public** **void** setBookname(String bookname)
35. {
36. **this**.bookname = bookname;
37. }
38. **public** String getAuthor()
39. {
40. **return** author;
41. }
42. **public** **void** setAuthor(String author)
43. {
44. **this**.author = author;
45. }
46. **public** **int** getPrice()
47. {
48. **return** price;
49. }
50. **public** **void** setPrice(**int** price)
51. {
52. **this**.price = price;
53. }
54. }

**Step 11:** Create a package with the name **com.javatpoint.controller**in the folder **src/main/java.**

**Step 12:** Create a Controller class in the package **com.javatpoint.controller.**We have created a controller class with the name **BooksController.**In the BooksController class, we have done the following:

* Mark the class as **RestController**by using the annotation**@RestController.**
* Autowire the **BooksService** class by using the annotation **@Autowired**.
* Define the following methods:
  + **getAllBooks():** It returns a List of all Books.
  + **getBooks():** It returns a book detail that we have specified in the path variable. We have passed bookid as an argument by using the annotation @PathVariable. The annotation indicates that a method parameter should be bound to a URI template variable.
  + **deleteBook():** It deletes a specific book that we have specified in the path variable.
  + **saveBook():** It saves the book detail. The annotation @RequestBody indicates that a method parameter should be bound to the body of the web request.
  + **update():** The method updates a record. We must specify the record in the body, which we want to update. To achieve the same, we have used the annotation @RequestBody.

**BooksController.java**

1. **package** com.javatpoint.controller;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.DeleteMapping;
5. **import** org.springframework.web.bind.annotation.GetMapping;
6. **import** org.springframework.web.bind.annotation.PathVariable;
7. **import** org.springframework.web.bind.annotation.PostMapping;
8. **import** org.springframework.web.bind.annotation.PutMapping;
9. **import** org.springframework.web.bind.annotation.RequestBody;
10. **import** org.springframework.web.bind.annotation.RestController;
11. **import** com.javatpoint.model.Books;
12. **import** com.javatpoint.service.BooksService;
13. //mark class as Controller
14. @RestController
15. **public** **class** BooksController
16. {
17. //autowire the BooksService class
18. @Autowired
19. BooksService booksService;
20. //creating a get mapping that retrieves all the books detail from the database
21. @GetMapping("/book")
22. **private** List<Books> getAllBooks()
23. {
24. **return** booksService.getAllBooks();
25. }
26. //creating a get mapping that retrieves the detail of a specific book
27. @GetMapping("/book/{bookid}")
28. **private** Books getBooks(@PathVariable("bookid") **int** bookid)
29. {
30. **return** booksService.getBooksById(bookid);
31. }
32. //creating a delete mapping that deletes a specified book
33. @DeleteMapping("/book/{bookid}")
34. **private** **void** deleteBook(@PathVariable("bookid") **int** bookid)
35. {
36. booksService.delete(bookid);
37. }
38. //creating post mapping that post the book detail in the database
39. @PostMapping("/books")
40. **private** **int** saveBook(@RequestBody Books books)
41. {
42. booksService.saveOrUpdate(books);
43. **return** books.getBookid();
44. }
45. //creating put mapping that updates the book detail
46. @PutMapping("/books")
47. **private** Books update(@RequestBody Books books)
48. {
49. booksService.saveOrUpdate(books);
50. **return** books;
51. }
52. }

**Step 13:** Create a package with the name **com.javatpoint.service**in the folder **src/main/java.**

**Step 14:** Create a **Service** class. We have created a service class with the name **BooksService**in the package **com.javatpoint.service.**

**BooksService.java**

1. **package** com.javatpoint.service;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.stereotype.Service;
6. **import** com.javatpoint.model.Books;
7. **import** com.javatpoint.repository.BooksRepository;
8. //defining the business logic
9. @Service
10. **public** **class** BooksService
11. {
12. @Autowired
13. BooksRepository booksRepository;
14. //getting all books record by using the method findaAll() of CrudRepository
15. **public** List<Books> getAllBooks()
16. {
17. List<Books> books = **new** ArrayList<Books>();
18. booksRepository.findAll().forEach(books1 -> books.add(books1));
19. **return** books;
20. }
21. //getting a specific record by using the method findById() of CrudRepository
22. **public** Books getBooksById(**int** id)
23. {
24. **return** booksRepository.findById(id).get();
25. }
26. //saving a specific record by using the method save() of CrudRepository
27. **public** **void** saveOrUpdate(Books books)
28. {
29. booksRepository.save(books);
30. }
31. //deleting a specific record by using the method deleteById() of CrudRepository
32. **public** **void** delete(**int** id)
33. {
34. booksRepository.deleteById(id);
35. }
36. //updating a record
37. **public** **void** update(Books books, **int** bookid)
38. {
39. booksRepository.save(books);
40. }
41. }

**Step 15:** Create a package with the name **com.javatpoint.repository**in the folder **src/main/java.**

**Step 16:** Create a **Repository** interface. We have created a repository interface with the name **BooksRepository**in the package **com.javatpoint.repository.**It extends the **Crud Repository** interface.

**BooksRepository.java**

1. **package** com.javatpoint.repository;
2. **import** org.springframework.data.repository.CrudRepository;
3. **import** com.javatpoint.model.Books;
4. //repository that extends CrudRepository
5. **public** **interface** BooksRepository **extends** CrudRepository<Books, Integer>
6. {
7. }

Now we will configure the datasource **URL, driver class name, username,**and**password,**in the **application.properties**file.

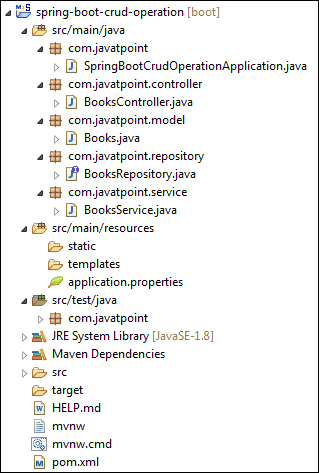
**Step 17:** Open the **application.properties** file and configure the following properties.

**application.properties**

1. spring.datasource.url=jdbc:h2:mem:books\_data
2. spring.datasource.driverClassName=org.h2.Driver
3. spring.datasource.username=sa
4. spring.datasource.password=
5. spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
6. #enabling the H2 console
7. spring.h2.console.enabled=**true**

#### **Note: Do not forget to enable the H2 console.**

After creating all the classes and packages, the project directory looks like the following.



Now we will run the application.

**Step 18:** Open **SpringBootCrudOperationApplication.java**file and run it as Java Application.

**SpringBootCrudOperationApplication.java**

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

#### **Note: In the next steps we will use rest client Postman. So, ensure that the Postman application is already installed in your system.**

**Step 19:** Open the **Postman**and do the following:

* Select the **POST**
* Invoke the URL http://localhost:8080/books.
* Select the **Body**
* Select he Content-Type **JSON (application/json).**
* Insert the data. We have inserted the following data in the Body:

1. {
2. "bookid": "5433",
3. "bookname": "Core and Advance Java",
4. "author": "R. Nageswara Rao",
5. "price": "800"
6. }

* Click on the **Send**

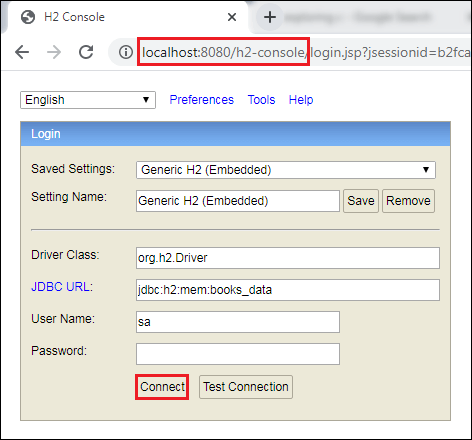
When the request is successfully executed, it shows the **Status:200 OK**. It means the record has been successfully inserted in the database.

Similarly, we have inserted the following data.

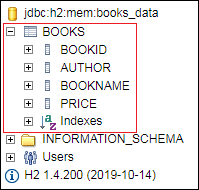
1. {
2. "bookid": "0982",
3. "bookname": "Programming with Java",
4. "author": "E. Balagurusamy",
5. "price": "350"
6. }
7. {
8. "bookid": "6321",
9. "bookname": "Data Structures and Algorithms in Java",
10. "author": "Robert Lafore",
11. "price": "590"
12. }
13. {
14. "bookid": "5433",
15. "bookname": "Effective Java",
16. "author": "Joshua Bloch",
17. "price": "670"
18. }

Let's access the H2 console to see the data.

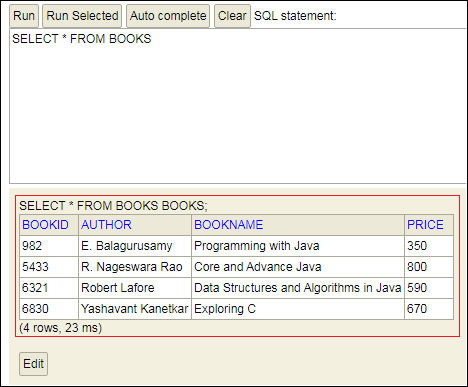
**Step 20:** Open the browser and invoke the URL http://localhost:8080/h2-console. Click on the **Connect** button, as shown below.



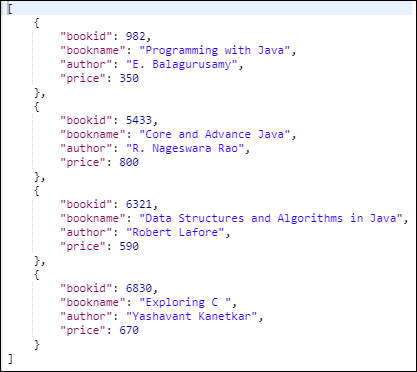
After clicking on the **Connect** button, we see the **Books** table in the database, as shown below.



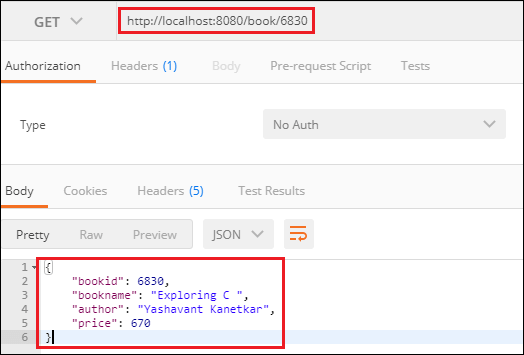
**Step 21:** Click on the **Books** table and then click on the **Run** button. The table shows the data that we have inserted in the body.



**Step 22:** Open the **Postman** and send a **GET** request with the URL http://localhost:8080/books. It returns the data that we have inserted in the database.

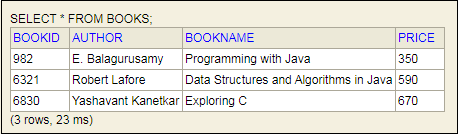


Let's send a **GET** request with the URL http://localhost:8080/book/{bookid}. We have specified the **bookid 6830**. It returns the detail of the book whose id is 6830.



Similarly, we can also send a **DELETE** request to delete a record. Suppose we want to delete a book record whose id is **5433**.

Select the **DELETE** method and invoke the URL http://localhost:8080/book/5433. Again, execute the **Select** query in the H2 console. We see that the book whose id is **5433** has been deleted from the database.



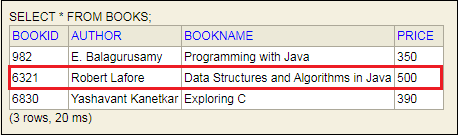
Similarly, we can also update a record by sending a **PUT** request. Let's update the price of the book whose id is **6321**.

* Select the **PUT**
* In the request body, paste the record which you want to update and make the changes. In our case, we want to update the record of the book whose id is 6321. In the following record, we have changed the price of the book.

1. {
2. "bookid": "6321",
3. "bookname": "Data Structures and Algorithms in Java",
4. "author": "Robert Lafore",
5. "price": "500"
6. }

* Click on the **Send**

Now, move to the H2 console and see the changes have reflected or not. We see that the price of the book has been changed, as shown below.



# Spring Boot Thymeleaf

## What is Thymeleaf?

The**Thymeleaf** is an open-source Java library that is licensed under the **Apache License 2.0**. It is a **HTML5/XHTML/XML** template engine. It is a **server-side Java template**engine for both web (servlet-based) and non-web (offline) environments. It is perfect for modern-day HTML5 JVM web development. It provides full integration with Spring Framework.

It applies a set of transformations to template files in order to display data or text produced by the application. It is appropriate for serving XHTML/HTML5 in web applications.

The goal of Thymeleaf is to provide a **stylish** and **well-formed**way of creating templates. It is based on XML tags and attributes. These XML tags define the execution of predefined logic on the DOM (Document Object Model) instead of explicitly writing that logic as code inside the template. It is a substitute for **JSP**.

The architecture of Thymeleaf allows the **fast** **processing** of templates that depends on the caching of parsed files. It uses the least possible amount of I/O operations during execution.

31M

675

C++ vs Java

## Why we use Thymeleaf?

JSP is more or less similar to HTML. But it is not completely compatible with HTML like Thymeleaf. We can open and display a Thymeleaf template file normally in the browser while the JSP file does not.

Thymeleaf supports variable expressions (${...}) like Spring EL and executes on model attributes, asterisk expressions (\*{...}) execute on the form backing bean, hash expressions (#{...}) are for internationalization, and link expressions (@{...}) rewrite URLs.

Like JSP, Thymeleaf works well for Rich HTML emails.

## What kind of templates can the Thymeleaf process?

Thymeleaf can process six types of templates (also known as **Template Mode**) are as follows:

* XML
* Valid XML
* XHTML
* Valid XHTML
* HTML5
* Legacy HTML5

Except for the Legacy HTML5 mode, all the above modes refer to **well-defined XML** files. It allows us to process HTML5 files with features such as **standalone tags, tag attributes without value, or not written between quotes**.

To process files in this specific mode, Thymeleaf performs a transformation that converts files into a **well-formed XML**file (valid HTML5 file).

#### **Note: In Thymeleaf, validation is available only for XHTML and XML template.**

Thymeleaf also allows us to define our own mode by specifying both a way to parse templates in this mode. In this way, whatever that can be modeled as a DOM tree could effectively be processed as a template by Thymeleaf.

## The Standard Dialect

Thymeleaf is a template engine framework that allows us to define the DOM nodes. The DOM nodes processed in the templates.

An object that applies logic to a DOM node is called **processor**. A set of processors, along with some extra artifacts, is called the **dialect**. The dialect that contains the Thymeleaf's core library is called the **Standard Dialect.**

If we want to define our own processing logic while taking advantage of the library's advanced features, we can define our own dialects. In a template engine, we can configure several dialects at a time.

The Thymeleaf integration packages (thymeleaf-spring3 and thymeleaf-spring4) define a dialect called **SpringStandard Dialect**. The Standard Dialect and SpringStandard are almost the same. But the Standard Dialect has some small changes that make better use of some features in Spring Framework.

For example, use Spring Expression Language instead of Thymeleaf's standard ONGL (Object-Graph Navigation Language).

The Standard Dialect can process template in any mode. But it is perfect for web-oriented template modes (HTML5 and XHTML). It supports and validates the following XHTML specifications:

* XHTML 1.0 Transitional
* XHTML 1.0 Strict
* XHTML 1.0 Frameset
* XHTML 1.1.

Standard Dialect processor is the attribute processor that allows browsers to display HTML5/XHTML template files before being processed. It is because they ignore the additional attributes.

For example, when a JSP file uses tag library, it includes a fragment of code not displayable by a browser like:

1. **<form:inputText** name="student.Name" value="${student.name}" **/>**

The Thymeleaf Standard Dialect allows us to achieve the same functionality with the following code.

1. **<input** type="text" name=" student Name" value="Thomas" th:value="${student.name}" **/>**

The above code also allows us to define a **value** attribute in it (**Thomas**). The value will be displayed when the prototype is opened in the browser. The attribute will be substituted by the value resulting from the evaluation of **${student.name}** during the Thymeleaf processing of the template.

It allows a designer and developer to work on the same template file that reduces the efforts required to transform a static prototype into a working template file. It is known as **Natural Templating.**

## Thymeleaf Features

* It works on both web and non-web environments.
* Java template engine for HTML5/ XML/ XHTML.
* Its high-performance parsed template cache reduces I/O to the minimum.
* It can be used as a template engine framework if required.
* It supports several template modes: XML, XHTML, and HTML5.
* It allows developers to extend and create custom dialect.
* It is based on modular features sets called dialects.
* It supports internationalization.

## Thymeleaf Implementation

We can implement Thymeleaf template engine by adding **spring-boot-starter-thymeleaf** dependency in our application's pom.xml file. Spring Boot configures template engine to read template file from **/resource/templates.**

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-thymeleaf**</artifactId>**
4. **</dependency>**

## Spring Boot Thymeleaf Example

Let's create a Spring Boot application and implement Thymeleaf template.

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/" \t "_blank)

.

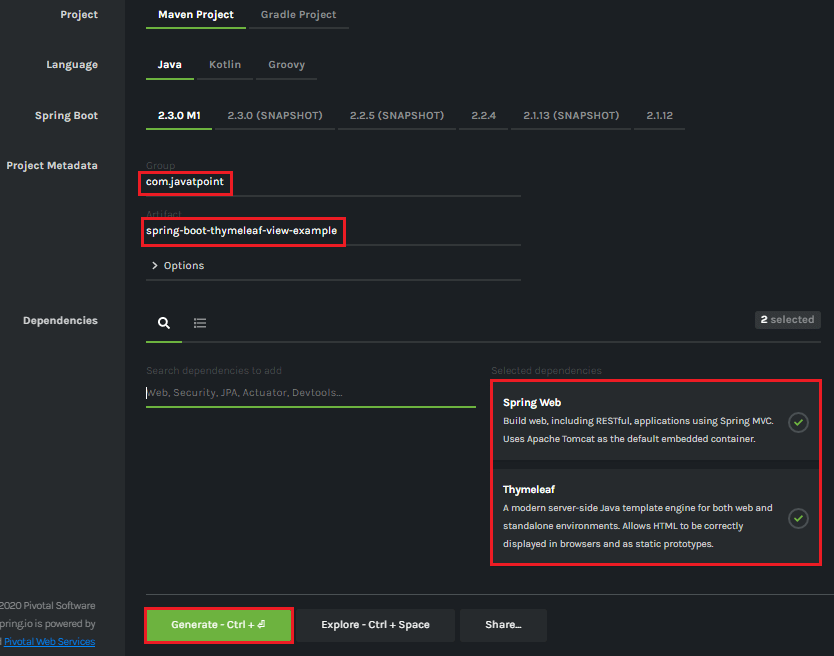
**Step 2:** Select the Spring Boot version **2.3.0.M1.**

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint**.

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-thymeleaf-view-example.**

**Step 5:** Add the dependencies **Spring Web** and **Thymeleaf**.

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the specifications in a **Jar** file and downloads it to the local system.



**Step 7: Extract** the Jar file and paste it into the STS workspace.

**Step 8: Import** the project folder in STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-thymeleaf-view-example -> Finish

It takes some time to import.

**Step 9:** Create a model class in the package **com.javatpoint**. We have created a model class with the name **User.**

In this class, we have defined two variables **name** and **email**and generate**Getters and Setters.**

**User.java**

1. **package** com.javatpoint;
2. **public** **class** User
3. {
4. String name;
5. String email;
6. **public** String getName()
7. {
8. **return** name;
9. }
10. **public** **void** setName(String name)
11. {
12. **this**.name = name;
13. }
14. **public** String getEmail()
15. {
16. **return** email;
17. }
18. **public** **void** setEmail(String email)
19. {
20. **this**.email = email;
21. }
22. }

**Step 10:** Create a controller class. We have created a controller class with the name **DemoController**.

**DemoController.java**

1. package com.javatpoint;
2. import org.springframework.web.bind.annotation.ModelAttribute;
3. import org.springframework.web.bind.annotation.RequestMapping;
4. import org.springframework.web.bind.annotation.RequestMethod;
5. import org.springframework.web.servlet.ModelAndView;
6. import org.springframework.stereotype.Controller;
7. @Controller
8. public class DemoController
9. {
10. @RequestMapping("/")
11. public String index()
12. {
13. return"index";
14. }
15. @RequestMapping(value="/save", method=RequestMethod.POST)
16. public ModelAndView save(@ModelAttribute User user)
17. {
18. ModelAndView modelAndView = new ModelAndView();
19. modelAndView.setViewName("user-data");
20. modelAndView.addObject("user", user);
21. return modelAndView;
22. }
23. }

In the next step, we will create the Thymeleaf templates.

**Step 11:** Inside the **templates** (src/main/resources/templates) folder of the project create a Thymeleaf template with the name **user-data**.

Right-click on the templates folder -> New -> Other -> HTML File -> Next -> Provide the File name -> Finish

#### **Note: Do not forget to implement the following in the template file.**

1. **<html** lang="en" xmlns:th="http://www.thymeleaf.org"**>**

**user-data.html**

1. **<html** xmlns:th="https://thymeleaf.org"**>**
2. **<table>**
3. **<tr>**
4. **<td><h4>**User Name: **</h4></td>**
5. **<td><h4** th:text="${user.name}"**></h4></td>**
6. **</tr>**
7. **<tr>**
8. **<td><h4>**Email ID: **</h4></td>**
9. **<td><h4** th:text="${user.email}"**></h4></td>**
10. **</tr>**
11. **</table>**
12. **</html>**

**Step 12:** Similarly, create an **HTML** file in the folder templates. We have created an HTML file with the name **index**.

**index.html**

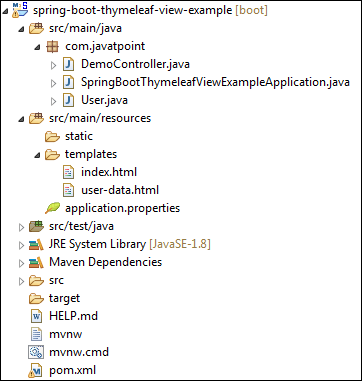
1. **<html** lang="en"**>**
2. **<head>**
3. **<title>**Index Page**</title>**
4. **</head>**
5. **<body>**
6. **<form** action="save" method="post"**>**
7. **<table>**
8. **<tr>**
9. **<td><label** for="user-name"**>**User Name**</label></td>**
10. **<td><input** type="text" name="name"**></input></td>**
11. **</tr>**
12. **<tr>**
13. **<td><label** for="email"**>**Email**</label></td>**
14. **<td><input** type="text" name="email"**></input></td>**
15. **</tr>**
16. **<tr>**
17. **<td></td>**
18. **<td><input** type="submit" value="Submit"**></input></td>**
19. **</tr>**
20. **</table>**
21. **</form>**
22. **</body>**
23. **</html>**

**Step 13:** Open the **application.properties** file and add the following properties in it.

**application.properties**

1. spring.thymeleaf.cache=false
2. spring.thymeleaf.suffix: .html

After creating all the files, folders, and packages, the project directory looks like the following:



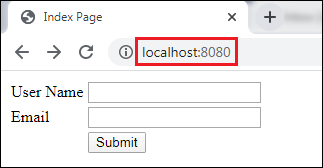
Let's run the application.

**Step 14:** Open the **SpringBootThymeleafViewExampleApplication.java**file and run it as Java Application.

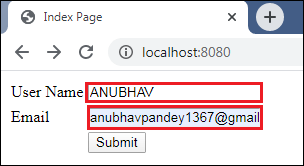
**SpringBootThymeleafViewExampleApplication.java**

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

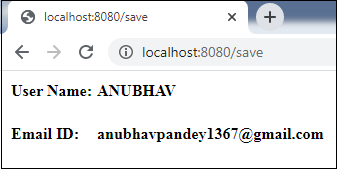
**Step 15:** Now, open the browser and invoke the URL http://localhost:8080. It shows the output, as shown below.



Provide the **User Name** and **Email** and click on the **Submit** button.



After clicking on the **Submit** button, the URL changes to http://localhost:8080/save and shows the show the user-data, as shown below.



In this section, we have discussed the Thymeleaf view. If we want to make the view more attractive, we can add **CSS** and **JS** files in the application. These files must be located under the **src/main/resources/static** folder.

# Spring Boot Caching

Spring Framework provides caching in a Spring Application, transparently. In Spring, the **cache abstraction** is a mechanism that allows consistent use of various caching methods with minimal impact on the code.

## Cache Abstraction

The cache abstraction mechanism applies to [**Java**](https://www.javatpoint.com/java-tutorial)**methods**. The main objective of using cache abstraction is to **reduce** the number of executions based on the information present in the cache. It applies to expensive methods such as**CPU**or**IO bound.**

Every time, when a method invokes, the abstraction applies a cache behavior to the method. It checks whether the method has already been executed for the given argument or not.

* If yes, the cached result is returned without executing the actual method.
* If no, first, the method executes, and the result is cached and returned to the user.

#### **Note: This approach works only for the methods that are guaranteed to return the same result for a given input. It does not matter how many times the method executes.**

The developers take care of two things while working with cache abstractions.

21.9M

415

Prime Ministers of India | List of Prime Minister of India (1947-2020)

* **Cache Declaration:** It identifies the methods that need to be cached.
* **Cache Configuration:** The backing cache where the data is stored and read from.

### **Caching**

Caching is a part of temporary memory ([RAM](https://www.javatpoint.com/ram-full-form)). It lies between the application and persistence database. It stores the recently used data that reduces the number of database hits as much as possible. In other words, caching is to store data for future reference.

## Why should we use the cache?

The primary reason for using cache is to make data access faster and less expensive. When the highly requested resource is requested multiple times, it is often beneficial for the developer to cache resources so that it can give responses quickly. Using cache in an application enhances the performance of the application. Data access from memory is always faster in comparison to fetching data from the database. It reduces both monetary cost and opportunity cost.

## What data should be cached?

* The data that do not change frequently.
* The frequently used read query in which results does not change in each call, at least for a period.

## Types of Caching

There are **four** types of caching are as follows:

* In-memory Caching
* Database Caching
* Web server Caching
* CDN Caching

### **In-memory Caching**

In-memory caching increases the performance of the application. It is the area that is frequently used. [**Memcached**](https://www.javatpoint.com/memcached-tutorial) and **Redis** are examples of in-memory caching. It stores key-value between application and database. Redis is an **in-memory, distributed,** and advanced caching tool that allows backup and restore facility. We can manage cache in distributed clusters, also.

### **Database Caching**

Database caching is a mechanism that generates web pages on-demand (dynamically) by fetching the data from the database. It is used in a **multi-tier** environment that involved clients, web-application server, and database. It improves **scalability** and **performance** by distributing a query workload. The most popular database caching is the first level cache of [Hibernate](https://www.javatpoint.com/hibernate-tutorial).

### **Web Server Caching**

Web server caching is a mechanism that stores data for **reuse**. For example, a copy of a web page served by a web server. It is cached for the first time when a user visits the page. If the user requests the same next time, the cache serves a copy of the page. It avoids server form getting overloaded. Web server caching enhances the page delivery speed and reduces the work to be done by the backend server.

### **CDN Caching**

The **CDN** stands for **Content Delivery Network**. It is a component used in modern web applications. It improves the delivery of the content by **replicating** commonly requested files (such as [HTML](https://www.javatpoint.com/html-tutorial) Pages, stylesheet, [JavaScript](https://www.javatpoint.com/javascript-tutorial), images, videos, etc.) across a globally distributed set of **caching servers.**

It is the reason CDN becomes more popular. The CDN reduces the load on an application origin and improves the user experience. It delivers a local copy of the content from a nearby **cache edge** (a cache server that is closer to the end-user), or a **Point of Presence (PoP)**.

## Cache vs. Buffer

|  |  |
| --- | --- |
| **Cache** | **Buffer** |
| The cache is based on **Least Recently Used**. | Buffer is based on **First-In-First-Out.** |
| It is the size of the page cache. | It is an in-memory raw block I/O buffer. |
| It lived for a **long** period. | It lived for a **short** period. |
| We **read** from the cache. | We **write** into the buffer. |
| It stores the **actual** file data. | It stores the file **metadata**. |
| It improves **read** performance. | It improves **write** performance. |

## Spring Boot Cache Annotations

### **@EnableCaching**

It is a class-level annotation. We can enable caching in the Spring Boot application by using the annotation **@EnableCaching.** It is defined in **org.springframework.cache.annotation**package. It is used together with **@Configuration**class.

The auto-configuration enables caching and setup a **CacheManager,** if there is no already defined instance of CacheManager. It scans for a specific provider, and when it does not find, it creates an in-memory cache using concurrent **HashMap.**

**Example**

In the following example, **@EnableCaching** annotation enables the cache mechanism.

1. @SpringBootApplication
2. @EnableCaching
3. **public** **class** SpringBootCachingApplication
4. {
5. **public** **static** **void** main(String[] args)
6. {
7. SpringApplication.run(SpringBootCachingApplication.**class**, args);
8. }
9. }

### **@CacheConfig**

It is a class-level annotation that provides a common cache-related setting. It tells the Spring where to store cache for the class. When we annotate a class with the annotation, it provides a set of default settings for any cache operation defined in that class. Using the annotation, we need not to declare things multiple times.

**Example**

In the following example, **employee** is the name of the cache.

1. @CacheConfig(cacheNames={"employee"})
2. **public** **class** UserService
3. {
4. //some code
5. }

### **@Caching**

It is used when we need both annotations **@CachePut**or **@CacheEvict**at the same time on the same method. In other words, it is used when we want to use multiple annotations of the same type.

But **Java does not allow multiple annotations** of the same type to be declared for a given method. To avoid this problem, we use**@Caching** annotation.

**Example**

In the following example, we have used the annotation **@Caching** and grouped all the **@CacheEvict** annotations.

1. @Caching(evict = {@CacheEvict("phone\_number"), @CacheEvict(value="directory", key="#student.id") })
2. **public** String getAddress(Student student)
3. {
4. //some code
5. }

### **@Cacheable**

It is a method level annotation. It defines a cache for a method's return value. The Spring Framework manages the requests and responses of the method to the cache that is specified in the annotation attribute. The @Cacheable annotation contains more options. For example, we can provide a **cache name** by using the **value** or **cacheNames** attribute.

We can also specify the **key** attribute of the annotation that uniquely identifies each entry in the cache. If we do not specify the key, Spring uses the default mechanism to create the key.

**Example**

In the following example, we have cached the **return value** of the method **studentInfo()** in **cacheStudentInfo,** and **id** is the unique key that identifies each entry in the cache.

1. @Cacheable(value="cacheStudentInfo", key="#id")
2. **public** List studentInfo()
3. {
4. //some code
5. **return** studentDetails;
6. }

We can also apply a condition in the annotation by using the condition attribute. When we apply the condition in the annotation, it is called **conditional caching**.

For example, the following method will be cached if the argument name has a length shorter than 20.

1. @Cacheable(value="student", condition="#name.length<20")
2. **public** Student findStudent(String name)
3. {
4. //some code
5. }

### **@CacheEvict**

It is a method level annotation. It is used when we want to remove stale or unused data from the cache. It requires one or multiple caches that are affected by the action. We can also specify a key or condition into it. If we want wide cache eviction, the @CacheEvict annotation provides a parameter called **allEntries**. It evicts all entries rather than one entry based on the key.

One important point about @CacheEvict annotation is that it can be used with void methods because the method acts as a trigger. It avoids return values. On the other hand, the annotation @Cacheable requires a return value that adds/updates data in the cache. We can use @CacheEvict annotation in the following ways:

Evict the whole cache:

1. @CacheEvict(allEntries=**true**)

Evict an entry by key:

1. @CacheEvict(key="#student.stud\_name")

**Example**

The following annotated method evicts all the data from the cache **student\_data**.

1. @CacheEvict(value="student\_data", allEntries=**true**) //removing all entries from the cache
2. **public** String getNames(Student student)
3. {
4. //some code
5. }

### **@CachePut**

It is a method level annotation. It is used when we want to **update** the cache without interfering the method execution. It means the method will always execute, and its result will be placed into the cache. It supports the attributes of @Cacheable annotation.

A point to be noticed that the annotations @Cacheable and @CachePut are not the same because they have different behavior. There is a slight difference between @Cacheable and @CachePut annotation is that the @**Cacheable** annotation **skips the method execution** while the **@CachePut** annotation **runs the method** and put the result into the cache.

**Example**

The following method will update the cache itself.

1. @CachePut(cacheNames="employee", key="#id") //updating cache
2. **public** Employee updateEmp(ID id, EmployeeData data)
3. {
4. //some code
5. }

## Spring Boot Cache Dependency

If we want to enable cache mechanism in a Spring Boot application, we need to add cache dependency in the pom.xml file. It enables caching and configures a CacheManager.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-cache**</artifactId>**
4. **</dependency>**

## Spring Boot Cache Example

Let's create a Spring Boot application and implement cache mechanism into it.

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

**Step 2:** Select the Spring Boot version **2.3.0.M1.**

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-cache-example.**

**Step 5:** Add the dependencies **Spring Web**and**Spring Cache Abstraction.**

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the specifications in a **Jar** file and downloads it to the local system.

**Step 7:** **Extract** the Jar file and paste it into the STS workspace.

**Step 8:** **Import** the project folder in STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-cache-example -> Finish

It takes some time to import.

Let's open the **pom.xml**file and see which dependencies we have added into it.

**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>**2.3.0.M1**</version>**
9. **<relativePath/>** <!-- lookup parent from repository -->
10. **</parent>**
11. **<groupId>**com.javatpoint**</groupId>**
12. **<artifactId>**spring-boot-cache-example**</artifactId>**
13. **<version>**0.0.1-SNAPSHOT**</version>**
14. **<name>**spring-boot-cache-example**</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-cache**</artifactId>**
23. **</dependency>**
24. **<dependency>**
25. **<groupId>**org.springframework.boot**</groupId>**
26. **<artifactId>**spring-boot-starter-web**</artifactId>**
27. **</dependency>**
28. **<dependency>**
29. **<groupId>**org.springframework.boot**</groupId>**
30. **<artifactId>**spring-boot-starter-test**</artifactId>**
31. **<scope>**test**</scope>**
32. **<exclusions>**
33. **<exclusion>**
34. **<groupId>**org.junit.vintage**</groupId>**
35. **<artifactId>**junit-vintage-engine**</artifactId>**
36. **</exclusion>**
37. **</exclusions>**
38. **</dependency>**
39. **</dependencies>**
40. **<build>**
41. **<plugins>**
42. **<plugin>**
43. **<groupId>**org.springframework.boot**</groupId>**
44. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
45. **</plugin>**
46. **</plugins>**
47. **</build>**
48. **<repositories>**
49. **<repository>**
50. **<id>**spring-milestones**</id>**
51. **<name>**Spring Milestones**</name>**
52. **<url>**https://repo.spring.io/milestone**</url>**
53. **</repository>**
54. **</repositories>**
55. **<pluginRepositories>**
56. **<pluginRepository>**
57. **<id>**spring-milestones**</id>**
58. **<name>**Spring Milestones**</name>**
59. **<url>**https://repo.spring.io/milestone**</url>**
60. **</pluginRepository>**
61. **</pluginRepositories>**
62. **</project>**

**Step 9:** Open the **SpringBootCacheExampleApplication.java** file and enable cache by adding the annotation **@EnableCaching.**

1. SpringBootCacheExampleApplication.java
2. **package** com.javatpoint;
3. **import** org.springframework.boot.SpringApplication;
4. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
5. **import** org.springframework.cache.annotation.EnableCaching;
6. @SpringBootApplication
7. //enabling caching
8. @EnableCaching
9. **public** **class** SpringBootCacheExampleApplication
10. {
11. **public** **static** **void** main(String[] args)
12. {
13. SpringApplication.run(SpringBootCacheExampleApplication.**class**, args);
14. }
15. }

**Step 10:**Create a package in the folder **src/main/java**with the name **com.javatpoint.model.**

**Step 11:**In the model package, create a class with the name **Customer** and define the following:

* Define three variables **accountno, customername, acounttype,** and **balance**.
* Generate **Constructor** **using fields**.  
  Right-click on the file -> Source -> Generate Constructor using Fields -> Select All -> Generate
* Generate **Getters and Setters.**  
  Right-click on the file -> Source -> Generate Getters and Setters -> Select All -> Generate

**Customer.java**

1. package com.javatpoint.model;
2. public class Customer
3. {
4. private int accountno;
5. private String customername;
6. private String accounttype;
7. private double balance;
8. public Customer(int accountno, String customername, String accounttype, double balance)
9. {
10. this.accountno = accountno;
11. this.customername = customername;
12. this.accounttype = accounttype;
13. this.balance = balance;
14. }
15. public int getAccountno()
16. {
17. return accountno;
18. }
19. public void setAccountno(int accountno)
20. {
21. this.accountno = accountno;
22. }
23. public String getCustomername()
24. {
25. return customername;
26. }
27. public void setCustomername(String customername)
28. {
29. this.customername = customername;
30. }
31. public String getAccounttype()
32. {
33. return accounttype;
34. }
35. public void setAccounttype(String accounttype)
36. {
37. this.accounttype = accounttype;
38. }
39. public double getBalance()
40. {
41. return balance;
42. }
43. public void setBalance(double balance)
44. {
45. this.balance = balance;
46. }
47. }

**Step 11:** Create a package in the folder **src/main/java**with the name **com.javatpoint.controller.**

**Step 12:** In the Controller package, create a controller class with the name **CustomerController**and do the following:

* Mark the class as **Controller** by using the annotation **@RestController.**
* Define a **mapping** for the controller by using the annotation **@RequestMapping.** We have defined the mapping **/customerinfo**.
* Create a **cache** for getting the data by using the annotation **@Cacheable.**We have defined the cache name by using the **value** attribute of the annotation.
* We have added two customer details in the

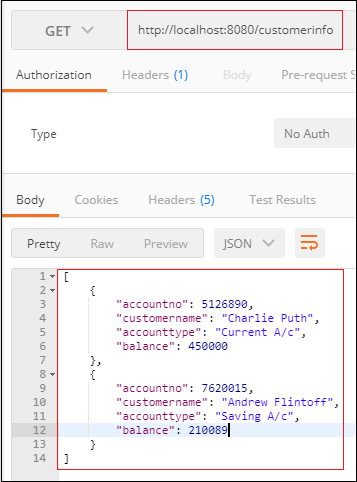
**CustomerController.java**

1. package com.javatpoint.controller;
2. import java.util.Arrays;
3. import java.util.List;
4. import org.springframework.cache.annotation.Cacheable;
5. import org.springframework.web.bind.annotation.RequestMapping;
6. import org.springframework.web.bind.annotation.RestController;
7. import com.javatpoint.model.Customer;
8. @RestController
9. public class CustomerController
10. {
11. @RequestMapping("/customerinfo")
12. //defines a cache for method's return value
13. @Cacheable(value="customerInfo")
14. public List customerInformation()
15. {
16. System.out.println("customer information from cache");
17. //adding customer detail in the List
18. List detail=Arrays.asList(new Customer(5126890,"Charlie Puth","Current A/c", 450000.00),
19. new Customer(7620015,"Andrew Flintoff","Saving A/c", 210089.00)
20. );
21. return detail;
22. }
23. }

Now run the application.

**Step 13:** Open the **SpringBootCacheExampleApplication.java**file and run it as Java Application.

**Step 14:** Open the **Postman** and send a **GET** request with the URL http://locahost:8080/custmerinfo. It returns the customer details, as shown below.



# Spring Boot Cache Provider

The Spring Boot framework allows the integration of various **cache providers,** such as **EhCache, Redis, Hazelcast, Infinispan, Caffeine,** etc. The cache provider allows the developer to configure cache transparently and explicitly in an application. We should use cache because it reduces the number of executions and increases the performance of the application.

In [Spring Boot](https://www.javatpoint.com/spring-boot-tutorial), the cache abstraction does not provide the actual space for the cache. It depends on the abstraction that occurred by the **org.springframework.cache.Cache** or **org.springframework.cache.CacheManager** interfaces.

## Caching Auto-configuration

The Spring Boot Framework simplifies the implementation of caching by auto-configuration support. It searches for the libraries and configuration-files in the classpath and initializes the required dependency beans at the time of application startup. The auto-configuration of caching includes the following steps:

* Add the annotation **@EnableCaching** in the configuration file.
* Add the required **caching libraries** in the classpath.
* In the root of the classpath, add the **configuration file**for the cache provider.

For example, if we want to implement **EhCache** in an application, first we enable the cache in the configuration file.

Keep Watching

1. @SpringBootApplication
2. @EnableCaching
3. **public** **class** Employee
4. {
5. @Bean
6. **public** CacheManager cacheManager()
7. {
8. //some code
9. }
10. }

Add **EhCache** dependency in the **pom.xml**file. It adds the required libraries in the classpath.

1. **<dependency>**
2. **<groupId>**org.ehcache**</groupId>**
3. **<artifactId>**ehcache**</artifactId>**
4. **</dependency>**

At the end, configure the file for cache provider. Here, we are using the EhCache so need to configure **ehcache.xml** file at the root of the classpath.

When we do not define a bean of type **CacheManager** or **CacheResolver**, the Spring Boot Framework tries to detect the following cache provider:

1. **Generic**
2. **JCache**
3. **EhCache**
4. **Hazelcast**
5. **Infinispan**
6. **Couchbase**
7. **Redis**
8. **Caffeine**
9. **Simple**

If the Spring Boot finds the more than one cache provider in the classpath, in such cases, we must specify the cache provider explicitly in the **application.properties**file.

1. spring.cache.ehcache.provider=net.sf.ehcache.CacheManager
2. spring.cache.ehcache.config=classpath:config/another-config.xml

We can set up a particular cache provider by using the property **spring.cache.type**. It is used in a certain environment if we want to disable caching.

1. spring.cache.type=none

The Spring Boot Framework provides a starter dependency that adds basic cache dependency in the application. The starter cache dependency, by default, provides the **spring-context-support** dependency.

#### **Note: We must include spring-context-support dependency in pom.xml file if we add cache dependency manually. Because, it provides support for Jcache, EhCache, and Caffiene.**

1. **<dependency>**
2. **<groupId>**org.springframework**</groupId>**
3. **<artifactId>**spring-context-support**</artifactId>**
4. **<version>**5.2.3.RELEASE**</version>**
5. **</dependency>**

The Spring Boot Framework automatically configures the CacheManager that can be further customized by implementing the **CacheManagerCustomizer** interface.

In the follwoig example, we have set up a flag that passes the null values to the primary map.

1. @Bean
2. **public** CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer()
3. {
4. **return** **new** CacheManagerCustomizer<ConcurrentMapCacheManager>()
5. {
6. @Override
7. **public** **void** customize(ConcurrentMapCacheManager cacheManager)
8. {
9. cacheManager.setAllowNullValues(**false**);
10. }
11. };
12. }

The above bean expects a auto-configured **ConcurrentMapCacheManager**. If the ConcurrentMapCacheManager is not auto-configures, the customizer will not invoke in any way. We can have any number of customizer and arrange them in order by using the annotation **@Order** or **@Ordered.**

### **Generic Caching**

If the spring-context-support defines at least one **org.springframework.cache.Cache**bean, it uses the Generic cache. The **CacheManager** bundled all the beans and configured them.

### **JCache**

JCache is a self-starting process that is provided by the **javax.cache.spi.CahingProvider.** It is present on the classpath JSR 107. The spring-boot-starter-cache provides the **JCacheCacheManager.**We can add any other cache library as well.

#### **Note: Spring Boot prefers the JSR support if a cache library provides both native implementation and JSR support.**

### **EhCache 2.x**

EHCache is Java-based, open-source, and widely used cache. In order to use EhCache we should use the following dependency.

1. **<dependency>**
2. **<groupId>**org.ehcache**</groupId>**
3. **<artifactId>**ehcache**</artifactId>**
4. **</dependency>**

There are two ways to configure EhCache:

* First, by configuring Java POJO file where all configuration parameters are configured through EhCache API.
* Second, by configuring the XML file where we configure EhCache according to the provided schema definition.

EhCache used a file called **ehcache.xml.**If the application found the file on the classpath, the **EhCacheCacheManager** provided by the spring-boot-starter-cache. We can configure the [XML](https://www.javatpoint.com/what-is-xml) file by using the following property:

1. spring.cache.ehcache.config=classpath:config/demo-config.xml

### **Hazelcast**

When we enable the caching in an application, Spring Boot wraps the **HazelcastInstance** automatically in the CacheManager. It distributes the data equally among the nodes. We can configure Hazelcast by using the following property.

1. spring.hazelcast.config=classpath:config/demo-hazelcast.xml

If the property is not set, Spring Boot tries to find the **hazelcast.xml**(Hazelcast configuration) file on the classpath.

### **Infinispan**

Infinispan is an embedded [Java](https://www.javatpoint.com/java-tutorial) library. It is used as a **cache** or a **data grid**. It stores data in the **key-value** form. It can be easily integrated with JCache, [JPA](https://www.javatpoint.com/jpa-tutorial) Quarkus, [Spring](https://www.javatpoint.com/spring-tutorial), etc.

It does not have a default file location, so we should specify it explicitly. If the infinispan is not specified explicitly, it uses default bootstrap.

1. spring.cache.infinispan.config=infinispan.xml

### **Couchbase**

The **CouchebaseCacheManager** is automatically configured when we implement **couchbase-spring-cache,** and Couchbase is configured. All the operations related to the cache perform in the **Bucket**. It allows us to create additional caches (if required) by setting up the property **spring.cache.cache-name.**

The customizer allows us to create additional Buckets, in which we can create another cache.

Let's understand the above concept through an example.

Suppose that we need three caches named **cacheA**, **cacheB,**and **cacheC**. The cacheA and cacheB are on the main Bucket (i.e., auto-configured Bucket). The cacheC is on another Bucket that is to live for a few seconds, say 4 seconds. Hence, we can create cacheA and cacheB by specifying the property, as follows:

1. spring.cache.cache-names=cacheA, cacheB

### **Redis**

The **RedisCacheManager** is autoconfigured when we configure **Redis**. It also allows us to create additional cache by using the property **spring.cache.cache-names**. The default configuration can be achieved by using the property **spring.cache.redis.\***.

We can take full control over the default configuration by using the **RedisCacheConfiguration**bean.

1. spring.cache.cache-names=cacheA, cacheB
2. spring.cache.redis.time-to-live=100000

The above properties configure two caches named cacheA and cacheB, that lives for 10 minutes.

### **Caffeine**

The caffeine is a Java based caching library. It also provides an in-memory cache. The spring-boot-starter-cache dependency automatically configures the **CaffeineCacheManger,** if it founds the Caffeine in the classpath. If we want to use Caffeine in an application, we need to add the following dependency:

1. **<dependency>**
2. **<groupId>**com.github.ben-manes.caffeine**</groupId>**
3. **<artifactId>**caffeine**</artifactId>**
4. **<version>**2.7.0**</version>**
5. **</dependency>**

The caffeine cache allows us to define **size** and **time to live** of the cache by using the property **spring.cache.caffeine.spec.**For example:

1. spring.cache.cache-names=cacheA,cacheB
2. spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s

The above configuration creates two caches named cache1 and cache2. The maximum size of the cache is **500** and the maximum time to live cache is **6** seconds.

### **Simple**

It is the default implementation. If no cache provider is specified. It configures a **ConcurrentHashMap** as a cache store if the Spring Boot does not find any cache provider in the classpath.

For example, if we want two caches, set their name by using the following property:

1. spring.cache.cache-names=cache1,cache2

### **None**

When we enable the cache by using the annotation @EnableCaching, the application expects a suitable configuration. It is used when we want to **disable** the cache in a certain environment. We use the property **spring.cache.type**to disable the cache.

1. spring.cache.type=none

# Spring Boot EhCaching

## EhCache

EhCache is an open-source, Java-based cache used to boost performance. The current version of Ehcache is **3**. It provides the implementation of the **JSR-107** cache manager. We can use it directly.

## Features of EhCache

* It is **fast**, **lightweight, Scalable,** and **Flexible**.
* It allows us to perform **Serializable** and **Object**
* It offers cache eviction policies such as **LRU, LFU, FIFO,**
* It stores the cache in **memory** and **disk**(SSD).
* It depends on **SLF4J** for logging.
* It has a full implementation of **JSR-107**and **Jcache**
* It supports distributed caching via **JGroups** or **JMS** and **RMI**.
* It uses **fluent query language**for distributed search.

## EhCache Usage Patterns

The cache uses several access patterns. There are following patterns used by EhCache:

* **Cache-aside**
* **Cache-as-SoR (system-of-record)**
* **Read-through**
* **Write-through**
* **Write-behind**

### **Cache-aside**

In the **cache-aside** pattern, first, the application consults with the cache. If the data is found, it returns the data directly. In the opposite scenario, it fetches the data from the SoR, stores it into the cache, and then return.

### **Cache-as-SoR**

The **cache-as-SoR**pattern represents SoR reading and writing operations to the cache. It reduces the responsibility of the application. It uses the combination of read and write pattern that includes**read-through, write-through,**and**write-behind.**It reduces the difficulty of the application. It allows the cache to solve the thundering-herd problem

26.1M

560

How to find Nth Highest Salary in SQL

### **Read-through**

The **read-through** pattern also copies the cache-aside pattern while reading data from the cache. The difference between the read-through and cache-aside is that read-through pattern implements the **CacheEntryFactory** interface. It guides the cache how to read an object from the cache. It is better to wrap the EhCache instance with the instance of **SelfPopulatingCache** while using the read-through pattern.

### **Write-through**

The **write-through** pattern also copies the cache-aside pattern while writing data in the cache. The difference between write-through and cache-aside pattern is that write-through pattern implements the **CacheWriter** interface. It configures the cache for both write-through and write-behind pattern. It writes data to the SoR in the same thread of execution.

### **Write-behind**

The **write-behind** pattern is different form the other three patterns. It modifies the cache entries after a **configurable delay**. The delay may in **seconds, minutes, a day, a week,** or for a**long time**. Simultaneously, it also queues the data to write at a later time in the same thread of execution.

The data write using write-behind pattern happens outside of the scope of the transaction. It means that it creates a new transaction to commit the data in the SoR that is distinct from the main transaction.

## EhCaching Storage Tiers

EhCache allows us to use various data storage areas, such as heap, disk and clustered. We can configure a multi-storage cache (uses more than one storage area). It can be arranged and managed as**tiers.**

The tiers are organized in order. The bottom-most tier is known as **authority tier,** and the other tier is known as the **caching tier**. It is also known as **nearer** or **near cache.**The caching tier can have more than one storage area. The hottest data kept in the caching tier because it is faster than the authority tier. Other data is kept in the authority tier that is slower but richer in comparison to the caching tier.

There are **four** types of data storage supported by EhCache:

* **On-Heap Store**
* **Off-Heap Store**
* **Disk Store**
* **Clustered Store**

### **On-Heap Store**

It stores cache entries in Java heap memory. It shares the storage with [Java](https://www.javatpoint.com/java-tutorial) application. It is fast because it uses heap but has limited storage space. The garbage collector also scans the on-heap store.

### **Off-Heap Store**

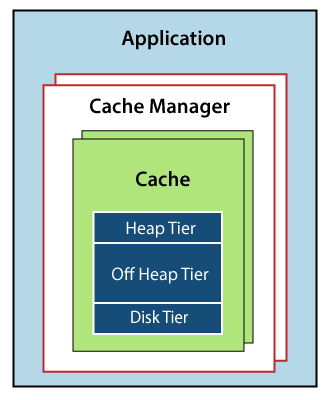
It uses the primary memory (RAM) to store cache entries. The garbage collector does not scan it. It is slower than the on-heap store because the cache entries move to the on-heap store before use. It is limited in size.

### **Disk Store**

It uses a disk to store cache entries. It is much slower than [RAM](https://www.javatpoint.com/ram-full-form)-based stores (on and off-heap store). It is better to use a dedicated disk if you are using a disk store pattern. It enhances throughput.

### **Clustered Store**

It stores cache entries on the remote server. It is slower than off-heap storage. It may have a failover server that provides high availability.



The above diagram shows that:

* An application may have more than one Cache Manager.
* Many caches can be handled by a Cache Manager.
* The caches can use more than one tier for storing cache entries.
* EhCache puts the recently used or frequently used data in the faster tier (caching tier).

## Configuring EhCache

* Place the **EhCache** jar in the classpath.
* Configure the **xml** and put it in the classpath.
* Create a
* Reference a Cache.

## Example of EhCache

In the following example, we are going to configure EhCache in an application.

**Step 1:** Open the [Spring](https://www.javatpoint.com/spring-tutorial) Initializr <https://start.spring.io/>.

**Step 2:** Select the [Spring Boot version](https://www.javatpoint.com/spring-boot-version) **2.3.0 M2**.

**Step 3:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

**Step 4:** Provide the **Artifact**. We have provided the Artifact **spring-boot-ehcache-example.**

**Step 5:** Add the **Spring Web** dependency.

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications related to application into a **Jar** file and downloads it to the local system.

**Step 7:** **Extract** the jar file.

**Step 8:** Copy the folder and paste it in the STS workspace.

**Step 9:** **Import** the project.

File -> Import -> Existing [Maven](https://www.javatpoint.com/maven-tutorial) Projects -> Next -> Browse -> Select the folder spring-boot-ehcache-example -> Select Folder -> Finish

It takes time to import the project.

**Step 10:**Copy the following dependency one by one from the [Maven](https://www.javatpoint.com/maven-repository) Repository <https://mvnrepository.com/> and paste it into the **pom.xml** file.

* **spring-boot-starter-cache**
* **ehcache 3**
* **cache API.**

#### **Note: Do not use the ehcache of the package net.sf.ehcache.**

**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 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**org.springframework.boot**</groupId>**
6. **<artifactId>**spring-boot-starter-parent**</artifactId>**
7. **<version>**2.3.0.M2**</version>**
8. **<relativePath/>** <!-- lookup parent from repository -->
9. **</parent>**
10. **<groupId>**com.javatpoint**</groupId>**
11. **<artifactId>**spring-boot-ehcache-example**</artifactId>**
12. **<version>**0.0.1-SNAPSHOT**</version>**
13. **<name>**spring-boot-ehcache-example**</name>**
14. **<description>**Demo project for Spring Boot**</description>**
15. **<properties>**
16. **<java.version>**1.8**</java.version>**
17. **</properties>**
18. **<dependencies>**
19. **<dependency>**
20. **<groupId>**org.springframework.boot**</groupId>**
21. **<artifactId>**spring-boot-starter-web**</artifactId>**
22. **</dependency>**
23. **<dependency>**
24. **<groupId>**org.springframework.boot**</groupId>**
25. **<artifactId>**spring-boot-starter-cache**</artifactId>**
26. **</dependency>**
27. **<dependency>**
28. **<groupId>**org.ehcache**</groupId>**
29. **<artifactId>**ehcache**</artifactId>**
30. **</dependency>**
31. **<dependency>**
32. **<groupId>**javax.cache**</groupId>**
33. **<artifactId>**cache-api**</artifactId>**
34. **</dependency>**
35. **<dependency>**
36. **<groupId>**org.springframework.boot**</groupId>**
37. **<artifactId>**spring-boot-starter-test**</artifactId>**
38. **<scope>**test**</scope>**
39. **<exclusions>**
40. **<exclusion>**
41. **<groupId>**org.junit.vintage**</groupId>**
42. **<artifactId>**junit-vintage-engine**</artifactId>**
43. **</exclusion>**
44. **</exclusions>**
45. **</dependency>**
46. **</dependencies>**
47. **<build>**
48. **<plugins>**
49. **<plugin>**
50. **<groupId>**org.springframework.boot**</groupId>**
51. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
52. **</plugin>**
53. **</plugins>**
54. **</build>**
55. **<repositories>**
56. **<repository>**
57. **<id>**spring-milestones**</id>**
58. **<name>**Spring Milestones**</name>**
59. **<url>**https://repo.spring.io/milestone**</url>**
60. **</repository>**
61. **</repositories>**
62. **<pluginRepositories>**
63. **<pluginRepository>**
64. **<id>**spring-milestones**</id>**
65. **<name>**Spring Milestones**</name>**
66. **<url>**https://repo.spring.io/milestone**</url>**
67. **</pluginRepository>**
68. **</pluginRepositories>**
69. **</project>**

Now, we need to configure the **ehcache.xml** file. It tells the framework where to find the file.

**Step 11:** Open the **application.properties** file and configure the EhCache by using the following property.

**application.properties**

1. #configuring ehcache.xml
2. spring.cache.jcache.config=classpath:ehcache.xml

**Step 12:** Open the **SpringBootEhcacheExampleApplication.java** file and enable caching by using the annotation **@EnableCaching**.

**SpringBootEhcacheExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cache.annotation.EnableCaching;
5. @SpringBootApplication
6. //enables the cache management capability
7. @EnableCaching
8. **public** **class** SpringBootEhcacheExampleApplication
9. {
10. **public** **static** **void** main(String[] args)
11. {
12. SpringApplication.run(SpringBootEhcacheExampleApplication.**class**, args);
13. }
14. }

#### **Note: If we do not want to use annotation @EnableCaching in main application file, we can create a separate CacheConfig class and annotate that call with the annotation.**

1. **package** com.javatpoint;
2. **import** org.springframework.cache.annotation.EnableCaching;
3. **import** org.springframework.context.annotation.Configuration;
4. @Configuration
5. //enable caching
6. @EnableCaching
7. **public** **class** CacheConfig
8. {
9. }

**Step 13:** Create a model class. We have created the model class in the package **com.javatpoint** with the name **Student.**In the model class do the following:

* Create five variable**id, name, gender,**and
* Generate **Constructor** using Fields.  
  Right-click on the file -> Source -> Generate Constructor using Fields -> Select All -> Generate
* Generate **Getters and Setters**.  
  Right-click on the file -> Source -> Generate Getters and Setters -> Select All -> Generate
* Generate a**toString()**
* Right-click on the file -> Source -> Generate toString() -> Generate

After completing all the above steps, the model class looks like the following.

**Student.java**

1. **package** com.javatpoint;
2. **public** **class** Student
3. {
4. **private** **int** id;
5. **private** String name;
6. **private** String gender;
7. **private** String city;
8. **public** Student(**int** id, String name, String gender, String city)
9. {
10. **super**();
11. **this**.id = id;
12. **this**.name = name;
13. **this**.gender = gender;
14. **this**.city = city;
15. }
16. **public** **int** getId()
17. {
18. **return** id;
19. }
20. **public** **void** setId(**int** id)
21. {
22. **this**.id = id;
23. }
24. **public** String getName()
25. {
26. **return** name;
27. }
28. **public** **void** setName(String name)
29. {
30. **this**.name = name;
31. }
32. **public** String getGender()
33. {
34. **return** gender;
35. }
36. **public** **void** setGender(String gender)
37. {
38. **this**.gender = gender;
39. }
40. **public** String getCity()
41. {
42. **return** city;
43. }
44. **public** **void** setCity(String city)
45. {
46. **this**.city = city;
47. }
48. @Override
49. **public** String toString()
50. {
51. **return** "Student [id=" + id + ", name=" + name + ", gender=" + gender + ", city=" + city + "]";
52. }
53. }

**Step 14:** Create a **Service** class that manages the student. We have created the service class with the name**StudentManager.**In this class, we have done the following:

* Annotated the class with the annotation **@Service.**
* Create an instance of **HashMap**.
* In the static block, we have added student data in the map.
* By using the annotation **@Cacheable,** we have defined the name of the cache All the data will be saved in this cache. We have defined the **id** in the **key** attribute of the annotation. The cache searches the student on the basis of **id**.
* We have created a method **getStudentById()** that parses id as a parameter. It returns the **id** of the student.

**StudentManager.java**

1. StudentManager.java
2. **package** com.javatpoint;
3. **import** java.util.HashMap;
4. **import** org.springframework.cache.annotation.Cacheable;
5. **import** org.springframework.stereotype.Service;
6. @Service
7. **public** **class** StudentManager
8. {
9. **static** HashMap<Integer, Student> student = **new** HashMap<>();
10. **static**
11. {
12. student.put(1, **new** Student(100, "Alex", "Male", "Berlin"));
13. student.put(2, **new** Student(101, "Tony", "Male", "Maxico"));
14. student.put(3, **new** Student(102, "Andrew", "Male", "Chicago"));
15. student.put(4, **new** Student(103, "Alexa", "Female", "Brussels"));
16. student.put(5, **new** Student(104, "Maria", "Female", "Houston"));
17. }
18. @Cacheable(cacheNames="demoCache", key="#id")
19. **public** Student getStudentById(Integer id)
20. {
21. System.out.println("Fetching student data from cache");
22. **return** student.get(id);
23. }
24. }

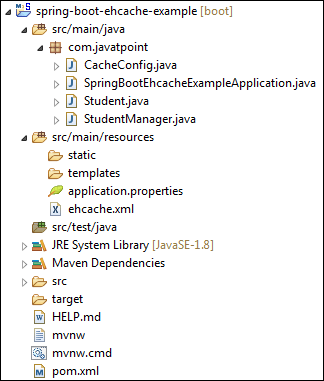
Now we need to create **ehcache.xml** file. It contains the information related to cache such as name of the cache, no of element in the memory, time to live data in the cache, etc.

**Step 15:** Create a cache configure file named **ehcache.xml** in the **src/main/resources** folder.

**ehcahe.xml**

1. <config
2. xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
3. xmlns='http://www.ehcache.org/v3'
4. xmlns:jsr107='http://www.ehcache.org/v3/jsr107'>
5. <ehcache>
6. <diskStore path="java.io.tmpdir" />
7. <defaultCache maxElementsInMemory="2000"
8. eternal="true"
9. overflowToDisk="false"
10. timeToLiveSeconds="1200" />
11. <cache name="demoCache"
12. maxElementsInMemory="2000"
13. eternal="false"
14. overflowToDisk="false"
15. timeToLiveSeconds="10000" />
16. </ehcache>
17. </config>

Now we have created all the required files. After creating all the files, the project directory looks like the following:



Let's run the application.

**Step 16:** Open the **SpringBootEhcacheExampleApplication.java** file and run it as [Java](https://www.javatpoint.com/java-tutorial) Application.

It shows the following output:

1. Getting Students from Cache
2. [id=100, name=Alex, gender=Male, city=Berlin]
3. [id=101, name=Tony, gender=Male, city=Mexico]
4. [id=102, name=Andrew, gender=Male, city=Chicago]
5. [id=103, name=Alexa, gender=Female, city=Brussels]
6. [id=104, name=Maria, gender=Female, city=Houston]

# How to Run Spring Boot Application

In this section, we will create and run a simple Spring Boot application.

## Creating a Spring Boot Application

**Step 1:** Open the Spring Initializr <https://start.spring.io/>.

**Step 2:** Select the [Spring Boot version](https://www.javatpoint.com/spring-boot-version) **2.2.2.BUILD-SNAPSHOT.**

**Step 3:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

30M

694

OOPs Concepts in Java

**Step 4:** Provide the **Artifact**. We have provided the Artifact **spring-boot-application-run.**

**Step 5:** Add the [**Spring**](https://www.javatpoint.com/spring-tutorial)**Web** dependency.

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications related to application into a **Jar** file and downloads it to the local system.

**Step 7:** **Extract** the jar file.

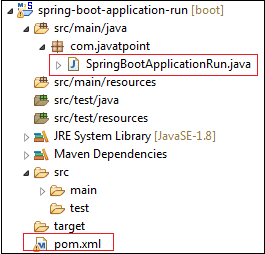
**Step 8:** Copy the folder and paste it in the STS workspace.

**Step 9:** **Import** the project.

File -> Import -> Existing Maven Projects -> Next -> Browse -> Select the folder spring- spring-boot-application-run -> Select Folder -> Finish

It takes time to import the project. When the project imports successfully, we can see it in the **Package Explorer**section of the IDE**.**

We see that two files are automatically created, one is **pom.xml,**and the other is **Application.java**file.



The **pom.xml** file contains the all the **dependencies**, **application name, Spring Boot version, group name, artifact,**and other **plugins.**

**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>**2.2.2.BUILD-SNAPSHOT**</version>**
9. **<relativePath/>** <!-- lookup parent from repository -->
10. **</parent>**
11. **<groupId>**com.javatpoint**</groupId>**
12. **<artifactId>**spring-boot-application-run**</artifactId>**
13. **<version>**0.0.1-SNAPSHOT**</version>**
14. **<name>**spring-boot-application-run**</name>**
15. **<description>**Demo project for Spring Boot**</description>**
16. **<properties>**
17. **<java.version>**1.8**</java.version>**
18. **</properties>**
20. **<dependencies>**
21. **<dependency>**
22. **<groupId>**org.springframework.boot**</groupId>**
23. **<artifactId>**spring-boot-starter**</artifactId>**
24. **</dependency>**
26. **<dependency>**
27. **<groupId>**org.springframework.boot**</groupId>**
28. **<artifactId>**spring-boot-starter-parent**</artifactId>**
29. **<version>**2.2.1.RELEASE**</version>**
30. **<type>**pom**</type>**
31. **</dependency>**
32. **<dependency>**
33. **<groupId>**org.springframework.boot**</groupId>**
34. **<artifactId>**spring-boot-starter-web**</artifactId>**
35. **</dependency>**
37. **<dependency>**
38. **<groupId>**org.springframework.boot**</groupId>**
39. **<artifactId>**spring-boot-starter-test**</artifactId>**
40. **<scope>**test**</scope>**
41. **<exclusions>**
42. **<exclusion>**
43. **<groupId>**org.junit.vintage**</groupId>**
44. **<artifactId>**junit-vintage-engine**</artifactId>**
45. **</exclusion>**
46. **</exclusions>**
47. **</dependency>**
48. **</dependencies>**
50. **<build>**
51. **<plugins>**
52. **<plugin>**
53. **<groupId>**org.springframework.boot**</groupId>**
54. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
55. **</plugin>**
56. **</plugins>**
57. **</build>**
59. **<repositories>**
60. **<repository>**
61. **<id>**spring-milestones**</id>**
62. **<name>**Spring Milestones**</name>**
63. **<url>**https://repo.spring.io/milestone**</url>**
64. **</repository>**
65. **<repository>**
66. **<id>**spring-snapshots**</id>**
67. **<name>**Spring Snapshots**</name>**
68. **<url>**https://repo.spring.io/snapshot**</url>**
69. **<snapshots>**
70. **<enabled>**true**</enabled>**
71. **</snapshots>**
72. **</repository>**
73. **</repositories>**
74. **<pluginRepositories>**
75. **<pluginRepository>**
76. **<id>**spring-milestones**</id>**
77. **<name>**Spring Milestones**</name>**
78. **<url>**https://repo.spring.io/milestone**</url>**
79. **</pluginRepository>**
80. **<pluginRepository>**
81. **<id>**spring-snapshots**</id>**
82. **<name>**Spring Snapshots**</name>**
83. **<url>**https://repo.spring.io/snapshot**</url>**
84. **<snapshots>**
85. **<enabled>**true**</enabled>**
86. **</snapshots>**
87. **</pluginRepository>**
88. **</pluginRepositories>**
90. **</project>**

The **main** class is a class that contains the main() method. It starts the Spring ApplicationContext. It is the class that we run for the execution of the application.

**SpringBootApplicationRun.java**

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

**Step 10:** Create a Controller. We have created a controller with the name **HelloWorldController**.

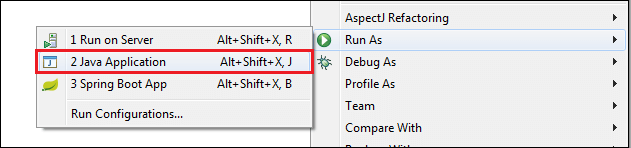
**HelloWorldController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.web.bind.annotation.RequestMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. @RestController
5. **public** **class** HelloWorldController
6. {
7. @RequestMapping("/")
8. **public** String hello()
9. {
10. **return** "Hello User";
11. }
12. }

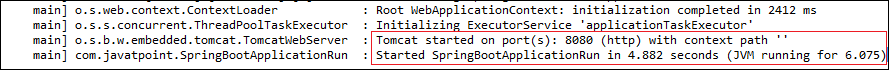
Now we have created all the required files related to [Spring Boot](https://www.javatpoint.com/spring-boot-tutorial) application.

## Run the Spring Boot application

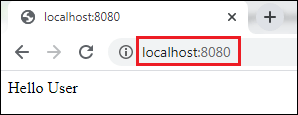
For running the [Spring Boot application](https://www.javatpoint.com/spring-boot-properties), open the main application file, and run it as **Java Application.**



When the application runs successfully, it shows the message in the console, as shown below.



Now, open the browser and invoke the URL http://localhost:8080. It shows the message that we have returned to the controller.



# Spring Boot Change Port

The Spring Boot framework provides the default embedded server (Tomcat) to run the Spring Boot application. It runs on port **8080**. It is possible to change the port in Spring Boot.

We can change the port in [Spring Boot](https://www.javatpoint.com/spring-boot-tutorial) by using the following interfaces and properties files:

* Using **application.properties** file
* Using **application.yml** file
* Using **EmbeddedServletContainerCustomizer** Interface
* Using **WebServerFactoryCustomizer** Interface
* Using **Command-Line Parameter**

## Using application.properties file

We recommend you to use the application.properties file if you want to change the default port. Because it is an easy and faster way to overwrite default values. We use the **server.port** property to overwrite the default property.

For example, if we want to change default port 8080 to 8082, specify the property in application.properties file.

How to find Nth Highest Salary in SQL

**application.properties**

1. server.port=8082

We can also set the port property to 0. It scans the random port for the application. It uses a new port whenever we restart our application.

**application.properties**

1. server.port=0

## Using application.yml file

Similarly, we can also change the default port by using a **yml** file. Use either **application.properties** or **application.yml** file, both the files works in the same manner.

**application.yml**

1. server:
2. port:8082

## Using EmbeddedServletContainerCustomizer Interface

If you are using **Spring Boot 1.x** version, it provides an interface **EmbeddedServletContainerCustomizer** to change the default port.

**EmbeddedServletContainerCustomizer Interface**

By using the EmbeddedServletContainerCustomizer, we can customize auto-configured embedded servlet containers. All the beans of this type get a callback with the container factory before starting the container itself. Therefore, we can set the **port, addresses**, and **even error pages**. It is defined in the **org.springframework.boot.context.embedded** package.

The interface contains a method called **customize()**. It allows us to customize and specify **ConfigurableEmbeddedServletContainer**. It parses a parameter called **container** that we want to customize.

### **Syntax**

1. void customize(ConfigurableEmbeddedServletContainer container)

**ConfigurableEmbeddedServletContainer Interface**

It is an interface that reflects the changes in the **EmbeddedServletContainerFactory** interface (factory interface used to create **EmbeddedServletContainers)**. It is defined in the **org.springframework.boot.context.embedded** package. It contains a method to change the port called the setPort() method.

**setPort() method**

The setPort() method configures the port of embedded servlet container should listen on. When we do not specify the port, it uses the default port **8080**. If we want to disable the auto-start feature of the embedded server, use the port **-1**. The port -1 represents that it will not listen to any port but start the web application context. The method parses a parameter port (the **por**t to set) of type int.

### **Syntax**

1. **void** setPort(**int** port)

In the following example, we have created a class named **ServerCustomizer** and implements the EmbeddedServletContainerCustomizer Interface. We have overridden the customize() method and invoke the setPort() method that sets the port **8097**.

**ServerCustomizer.java**

1. @Component
2. **public** **class** ServerCustomizer **implements** EmbeddedServletContainerCustomizer
3. {
4. @Override
5. **public** **void** customize(ConfigurableEmbeddedServletContainer container)
6. {
7. container.setPort(8097);
8. }
9. }

## Using WebServerFactoryCustomizer Interface

Spring Boot 2.x version provides **WebServerFactoryCustomizer** interface to change the default port. It defined in the package **org.springframework.boot.web.server**. It parses a parameter **T** of type web server factory.

### **Syntax**

1. **public** **interface** WebServerFactoryCustomizer<T **extends** WebServerFactory<

The interface contains a method called **customize()**. It allows us to customize web server factories. It parses a parameter called **factory** that we want to customize. All the beans of this type get a callback with the server factory before starting the container itself. Therefore, we can set the **port, addresses**, and **even error pages**.

### **Syntax**

1. **void**  customize(T factory)

**WebServerFactory Interface**

It is a tagging interface for factories. It is defined in **org.springframework.boot.web.server** package. It creates a **WebServer**.

**ConfigurableWebServerFactory**

It is an interface that configures web server factory. It is defined in the package **org.springframework.boot.web.server**. It extends **WebServerFactory** and **ErrorPageRegistory**. It contains a method to change the port called the **setPort()** method.

**setPort()**

The setPort() method configures the port of embedded servlet container should listen on. When we do not specify the port, it uses the default port **8080**. If we want to disable the auto-start feature of the embedded server, use the port **-1**. The port -1 represents that it will not listen to any port but start the web application context. The method parses a parameter port (the **port** to set) of type int.

### **Syntax**

1. **void** setPort(**int** port)

#### **Note: This setPort() method is of interface ConfigurableWebServerFactory**

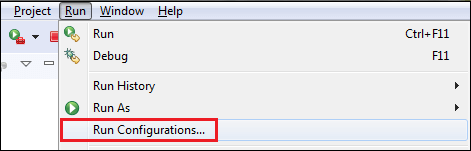
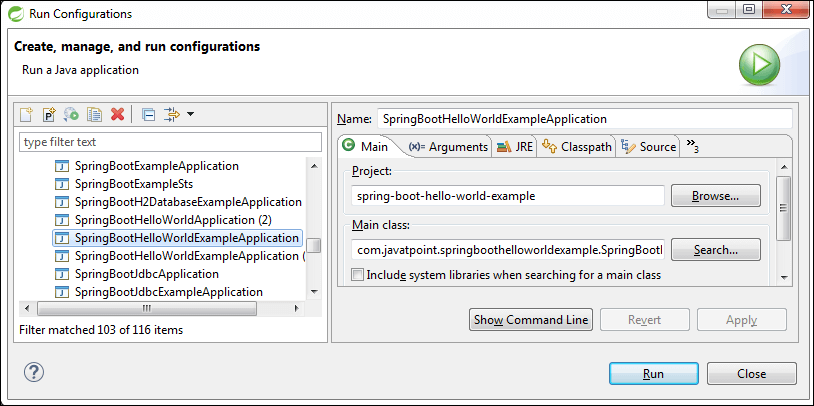
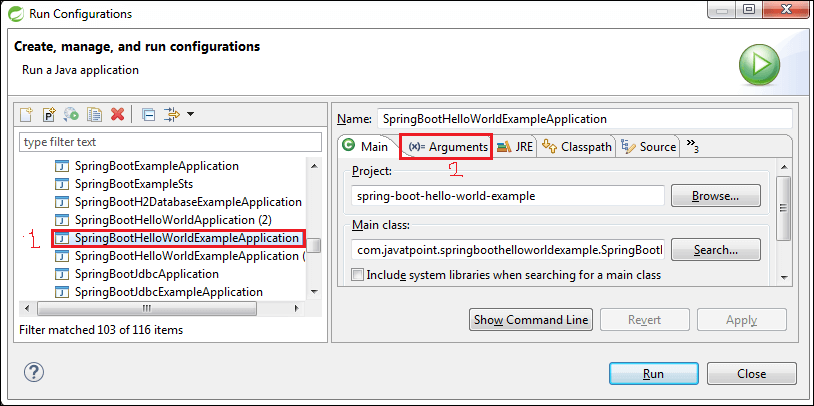
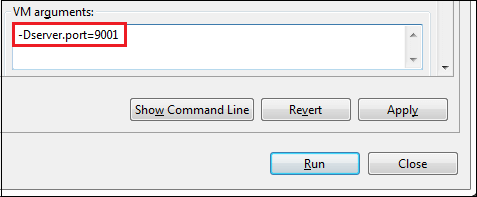
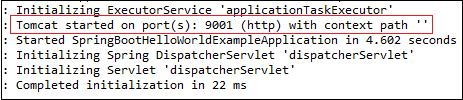
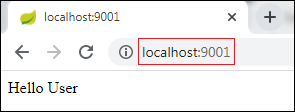
In the following example, we have created a class named **ServerCustomizer** that implements the WebServerFactoryCustomizer Interface. We have overridden the customize() method and invoke the setPort() method that sets the port **9001**.

**ServerCustomizer.java**

1. @Component
2. **public** **class** ServerCustomizer **implements** WebServerFactoryCustomizer< ConfigurableWebServerFactory <
3. {
4. @Override
5. **public** **void** customize(ConfigurableWebServerFactory factory)
6. {
7. factory.setPort(9001);
8. }
9. }

## Using Command Line Parameter

We can also change the port in Spring Boot by using the command line parameter. We must follow the steps given below:

* Open any Spring Boot application.
* Click on **Run** menu and select **Run Configurations** Or right-click on the application file -< Run As -< **Run Configurations**. Run Configurations window appears on the screen.  
    
  **Run Configurations window** appears on the screen.  
  
* Select the application file in which you want to change the port. In our case, we want to change the port of **SpringBootHelloWorldExampleApplication**, so we have selected it.
* Click on the **Arguments** tab.  
  
* Write **-Dserver.port=9001** in the **VM arguments** field. You can specify your own port instead of 9001.  
  
* Now, click on **Apply** and **Run** button, respectively.  
  After clicking on the Run button, the application starts running. We can see the console to see on which port server is running, as shown below.  
  
* Open the browser and invoke the URL http://localhost:9001. It runs the application on port **9001**.  
  

# Spring Boot REST Example

The REST application follows the REST architectural approach. We use the REST application for developing and designing networked applications. It generates the HTTP request that performs CRUD operations on the data. Usually, it returns data in JSON or [XML](https://www.javatpoint.com/xml-tutorial) format.

## Spring Boot REST API Example

In the following example, we are going to create a REST application. In this application, we have created a list of products and return the same list. It returns the data in [JSON](https://www.javatpoint.com/json-tutorial) format.

Let's implement it in RSET application and understand the REST approach by using the following steps.

**Step 1:** Open the [Spring](https://www.javatpoint.com/spring-tutorial) Initializr <https://start.spring.io/>.

2:00:00/04:28

10 Sec

33M

569

Hello Java Program for Beginners

**Step 2:** Select the [Spring Boot version](https://www.javatpoint.com/spring-boot-version) **2.3.0.M2**.

**Step 3:** Provide the **Group** name. We have provided the Group name **com.javatpoint**.

**Step 4:** Provide the **Artifact**. We have provided the **Artifact spring-boot-rest-example**.

**Step 5:** Add the **Spring Web** dependency.

**Step 6:** Click on the Generate button. When we click on the **Generate** button, it wraps all the specifications related to application into a **Jar** file and downloads it to the local system.

**Step 7:** **Extract** the jar file.

**Step 8:** Copy the folder and paste it in the STS workspace.

**Step 9:** **Import** the project.

File -> Import -> Existing Maven Projects -> Next -> Browse -> Select the folder spring- spring-boot-rest-example -> Select Folder -> Finish

It takes time to import the project. When the project imports successfully, we can see it in the **Package Explorer** section of the IDE.

**Step 10:** Create a **model** class in the package **com.javatpoint**. We have created a model class with the name **Product**. In this class, do the following:

* Create five variable id, pname, batchno, price, and noofproduct.
* Create a default constructor.
* Generate **Constructor using Fields**.  
  Right-click on the file -> Source -> Generate Constructor using Fields -> Select All -> Generate
* Generate **Getters** and Setters.

Right-click on the file -> Source -> Generate Getters and Setters -> Select All -> Generate

After completing all the steps, the model class looks like the following.

**Product.java**

1. **package** com.javatpoint;
2. **public** **class** Product
3. {
4. **private** **int** id;
5. **private** String pname;
6. **private** String batchno;
7. **private** **double** price;
8. **private** **int** noofproduct;
9. //default constructor
10. **public** Product()
11. {
13. }
14. //constructor using fields
15. **public** Product(**int** id, String pname, String batchno, **double** price, **int** noofproduct)
16. {
17. **super**();
18. **this**.id = id;
19. **this**.pname = pname;
20. **this**.batchno = batchno;
21. **this**.price = price;
22. **this**.noofproduct = noofproduct;
23. }
24. //getters and setters
25. **public** **int** getId()
26. {
27. **return** id;
28. }
29. **public** **void** setId(**int** id)
30. {
31. **this**.id = id;
32. }
33. **public** String getPname()
34. {
35. **return** pname;
36. }
37. **public** **void** setPname(String pname)
38. {
39. **this**.pname = pname;
40. }
41. **public** String getBatchno()
42. {
43. **return** batchno;
44. }
45. **public** **void** setBatchno(String batchno)
46. {
47. **this**.batchno = batchno;
48. }
49. **public** **double** getPrice()
50. {
51. **return** price;
52. }
53. **public** **void** setPrice(**double** price)
54. {
55. **this**.price = price;
56. }
57. **public** **int** getNoofproduct()
58. {
59. **return** noofproduct;
60. }
61. **public** **void** setNoofproduct(**int** noofproduct)
62. {
63. **this**.noofproduct = noofproduct;
64. }
65. }

Now, we need to create a controller.

**Step 11:** In the **com.javatpoint** package, create a Controller. We have created a controller with the name **ProductController**.

* Annotate the class with the annotation **@RestController**.
* We have autowired the **IProductService** interface. We will create it in the next step.
* We have created a mapping **/product** by using the annotation **@GetMapping**.
* We have mapped a method **getProduct()** to the **/product**. The method returns a list of products.

**ProductController.java**

1. **package** com.javatpoint;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.RestController;
6. @RestController
7. **public** **class** ProductController
8. {
9. @Autowired
10. **private** IProductService productService;
11. //mapping the getProduct() method to /product
12. @GetMapping(value = "/product")
13. **public** List<Product> getProduct()
14. {
15. //finds all the products
16. List<Product> products = productService.findAll();
17. //returns the product list
18. **return** products;
19. }
20. }

**Step 12:** Create an interface in the package **com.javatpoint** with the name **IProductService** and define the **findAll()** method that returns a List of products.

**IProductService.java**

1. **package** com.javatpoint;
2. **import** java.util.List;
3. **public** **interface** IProductService
4. {
5. List<Product> findAll();
6. }

**Step 13:** Create a **Service** class. We have created a service class in the package **com.javatpoint** with the name **ProductService**.

* Annotate the class with the annotation **@Service** and implements the **IProductService** interface.
* In this class, override the **findAll()** method by using the annotation **@Override**. The findAll() method of the ProductService class overrides the findAll() method of the **IProductService** interface.
* Create an object of **ArrayList**.
* **Add** the products in the ArrayList.
* Return the **list** of the products.

**ProductService.java**

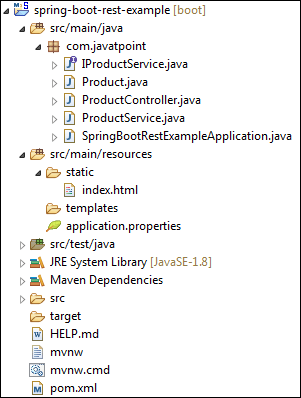
1. **package** com.javatpoint;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** org.springframework.stereotype.Service;
5. @Service
6. **public** **class** ProductService **implements** IProductService
7. {
8. @Override
9. **public** List<Product> findAll()
10. {
11. //creating an object of ArrayList
12. ArrayList<Product> products = **new** ArrayList<Product>();
13. //adding products to the List
14. products.add(**new** Product(100, "Mobile", "CLK98123", 9000.00, 6));
15. products.add(**new** Product(101, "Smart TV", "LGST09167", 60000.00, 3));
16. products.add(**new** Product(102, "Washing Machine", "38753BK9", 9000.00, 7));
17. products.add(**new** Product(103, "Laptop", "LHP29OCP", 24000.00, 1));
18. products.add(**new** Product(104, "Air Conditioner", "ACLG66721", 30000.00, 5));
19. products.add(**new** Product(105, "Refrigerator ", "12WP9087", 10000.00, 4));
20. //returns a list of product
21. **return** products;
22. }
23. }

**Step 14:** In the **static** folder (src/main/resources/static), create an HTML file. We have created an [HTML](https://www.javatpoint.com/html-tutorial) file with the name **index**. In this file we have created a link of **Get all Products**.

**index.html**

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>Home page</title>
5. <meta charset="UTF-8">
6. <meta name="viewport" content="width=device-width, initial-scale=1.0">
7. </head>
8. <body>
9. <p>
10. <a href="product">Get all Products</a>
11. </p>
12. </body>
13. </html>

Now we have created all the files and folders. After creating all the files, the project directory looks like the fooling:



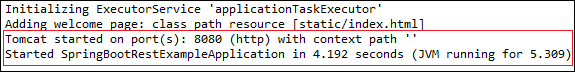
Let's run the application.

**Step 15:** Open the **SpringBootRestExampleApplication.java** file and run it as [Java](https://www.javatpoint.com/java-tutorial) Application. By default, it runs on port **8080**.

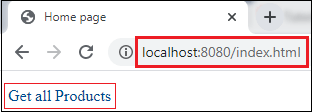
**SpringBootRestExampleApplication.java**

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

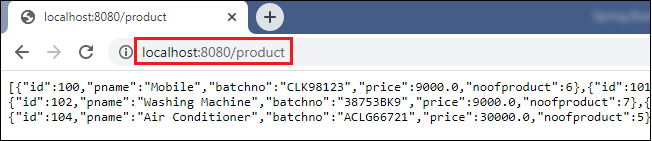
When the application runs successfully, it shows the message, as shown below



**Step 16:** Open the browser and invoke the [URL](https://www.javatpoint.com/url-full-form) http://localhost:8080/index.html. It shows the link of **Get all Products**, as shown in the following image.



Click on the link **Get all Products**. It returns a list of products in **JSON** format and the URL change to http://localhost:8080/product.



# Spring Boot REST Example

The REST application follows the REST architectural approach. We use the REST application for developing and designing networked applications. It generates the HTTP request that performs CRUD operations on the data. Usually, it returns data in JSON or [XML](https://www.javatpoint.com/xml-tutorial) format.

## Spring Boot REST API Example

In the following example, we are going to create a REST application. In this application, we have created a list of products and return the same list. It returns the data in [JSON](https://www.javatpoint.com/json-tutorial) format.

Let's implement it in RSET application and understand the REST approach by using the following steps.

**Step 1:** Open the [Spring](https://www.javatpoint.com/spring-tutorial) Initializr <https://start.spring.io/>.

31.4M

682

C++ vs Java

**Next**

**Stay**

**Step 2:** Select the [Spring Boot version](https://www.javatpoint.com/spring-boot-version) **2.3.0.M2**.

**Step 3:** Provide the **Group** name. We have provided the Group name **com.javatpoint**.

**Step 4:** Provide the **Artifact**. We have provided the **Artifact spring-boot-rest-example**.

**Step 5:** Add the **Spring Web** dependency.

**Step 6:** Click on the Generate button. When we click on the **Generate** button, it wraps all the specifications related to application into a **Jar** file and downloads it to the local system.

**Step 7:** **Extract** the jar file.

**Step 8:** Copy the folder and paste it in the STS workspace.

**Step 9:** **Import** the project.

File -> Import -> Existing Maven Projects -> Next -> Browse -> Select the folder spring- spring-boot-rest-example -> Select Folder -> Finish

It takes time to import the project. When the project imports successfully, we can see it in the **Package Explorer** section of the IDE.

**Step 10:** Create a **model** class in the package **com.javatpoint**. We have created a model class with the name **Product**. In this class, do the following:

* Create five variable id, pname, batchno, price, and noofproduct.
* Create a default constructor.
* Generate **Constructor using Fields**.  
  Right-click on the file -> Source -> Generate Constructor using Fields -> Select All -> Generate
* Generate **Getters** and Setters.

Right-click on the file -> Source -> Generate Getters and Setters -> Select All -> Generate

After completing all the steps, the model class looks like the following.

**Product.java**

1. **package** com.javatpoint;
2. **public** **class** Product
3. {
4. **private** **int** id;
5. **private** String pname;
6. **private** String batchno;
7. **private** **double** price;
8. **private** **int** noofproduct;
9. //default constructor
10. **public** Product()
11. {
13. }
14. //constructor using fields
15. **public** Product(**int** id, String pname, String batchno, **double** price, **int** noofproduct)
16. {
17. **super**();
18. **this**.id = id;
19. **this**.pname = pname;
20. **this**.batchno = batchno;
21. **this**.price = price;
22. **this**.noofproduct = noofproduct;
23. }
24. //getters and setters
25. **public** **int** getId()
26. {
27. **return** id;
28. }
29. **public** **void** setId(**int** id)
30. {
31. **this**.id = id;
32. }
33. **public** String getPname()
34. {
35. **return** pname;
36. }
37. **public** **void** setPname(String pname)
38. {
39. **this**.pname = pname;
40. }
41. **public** String getBatchno()
42. {
43. **return** batchno;
44. }
45. **public** **void** setBatchno(String batchno)
46. {
47. **this**.batchno = batchno;
48. }
49. **public** **double** getPrice()
50. {
51. **return** price;
52. }
53. **public** **void** setPrice(**double** price)
54. {
55. **this**.price = price;
56. }
57. **public** **int** getNoofproduct()
58. {
59. **return** noofproduct;
60. }
61. **public** **void** setNoofproduct(**int** noofproduct)
62. {
63. **this**.noofproduct = noofproduct;
64. }
65. }

Now, we need to create a controller.

**Step 11:** In the **com.javatpoint** package, create a Controller. We have created a controller with the name **ProductController**.

* Annotate the class with the annotation **@RestController**.
* We have autowired the **IProductService** interface. We will create it in the next step.
* We have created a mapping **/product** by using the annotation **@GetMapping**.
* We have mapped a method **getProduct()** to the **/product**. The method returns a list of products.

**ProductController.java**

1. **package** com.javatpoint;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.RestController;
6. @RestController
7. **public** **class** ProductController
8. {
9. @Autowired
10. **private** IProductService productService;
11. //mapping the getProduct() method to /product
12. @GetMapping(value = "/product")
13. **public** List<Product> getProduct()
14. {
15. //finds all the products
16. List<Product> products = productService.findAll();
17. //returns the product list
18. **return** products;
19. }
20. }

**Step 12:** Create an interface in the package **com.javatpoint** with the name **IProductService** and define the **findAll()** method that returns a List of products.

**IProductService.java**

1. **package** com.javatpoint;
2. **import** java.util.List;
3. **public** **interface** IProductService
4. {
5. List<Product> findAll();
6. }

**Step 13:** Create a **Service** class. We have created a service class in the package **com.javatpoint** with the name **ProductService**.

* Annotate the class with the annotation **@Service** and implements the **IProductService** interface.
* In this class, override the **findAll()** method by using the annotation **@Override**. The findAll() method of the ProductService class overrides the findAll() method of the **IProductService** interface.
* Create an object of **ArrayList**.
* **Add** the products in the ArrayList.
* Return the **list** of the products.

**ProductService.java**

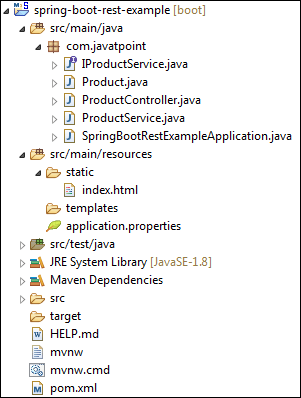
1. **package** com.javatpoint;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** org.springframework.stereotype.Service;
5. @Service
6. **public** **class** ProductService **implements** IProductService
7. {
8. @Override
9. **public** List<Product> findAll()
10. {
11. //creating an object of ArrayList
12. ArrayList<Product> products = **new** ArrayList<Product>();
13. //adding products to the List
14. products.add(**new** Product(100, "Mobile", "CLK98123", 9000.00, 6));
15. products.add(**new** Product(101, "Smart TV", "LGST09167", 60000.00, 3));
16. products.add(**new** Product(102, "Washing Machine", "38753BK9", 9000.00, 7));
17. products.add(**new** Product(103, "Laptop", "LHP29OCP", 24000.00, 1));
18. products.add(**new** Product(104, "Air Conditioner", "ACLG66721", 30000.00, 5));
19. products.add(**new** Product(105, "Refrigerator ", "12WP9087", 10000.00, 4));
20. //returns a list of product
21. **return** products;
22. }
23. }

**Step 14:** In the **static** folder (src/main/resources/static), create an HTML file. We have created an [HTML](https://www.javatpoint.com/html-tutorial) file with the name **index**. In this file we have created a link of **Get all Products**.

**index.html**

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>Home page</title>
5. <meta charset="UTF-8">
6. <meta name="viewport" content="width=device-width, initial-scale=1.0">
7. </head>
8. <body>
9. <p>
10. <a href="product">Get all Products</a>
11. </p>
12. </body>
13. </html>

Now we have created all the files and folders. After creating all the files, the project directory looks like the fooling:



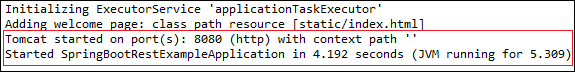
Let's run the application.

**Step 15:** Open the **SpringBootRestExampleApplication.java** file and run it as [Java](https://www.javatpoint.com/java-tutorial) Application. By default, it runs on port **8080**.

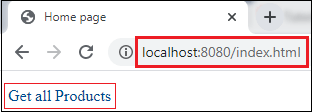
**SpringBootRestExampleApplication.java**

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

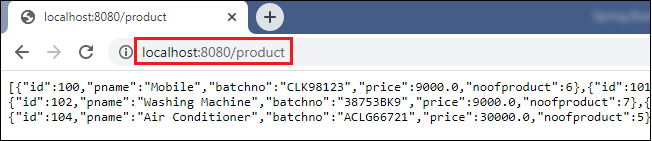
When the application runs successfully, it shows the message, as shown below



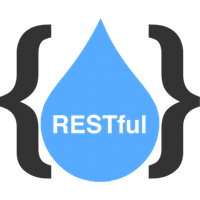
**Step 16:** Open the browser and invoke the [URL](https://www.javatpoint.com/url-full-form) http://localhost:8080/index.html. It shows the link of **Get all Products**, as shown in the following image.



Click on the link **Get all Products**. It returns a list of products in **JSON** format and the URL change to http://localhost:8080/product.



# RESTful Web Services Tutorial



RESTful Web Services are client and server applications that communicate over the WWW. RESTful Web Services are REST Architecture based Web Services. In REST Architecture, everything is a resource. RESTful Web Services provides communication between software applications running on different platforms and frameworks. We can consider web services as code on demand. A RESTful Web Service is a function or method which can be called by sending an HTTP request to a URL, and the service returns the result as the response. In this tutorial, you will learn the basics of RSETful Web Services with suitable examples and projects.

## Audience

Our RESTful Web Services tutorial is designed for Software Professionals and beginners who want to learn RESTful Web Services in easy steps. This tutorial will give you an in-depth knowledge of RESTful Web Services concepts. After completing this tutorial, you can develop RSESTful services.

## Prerequisites

Before continuing with this tutorial, you should have a basic understanding of Java, Spring, and Spring Boot Framework. Because we are going to develop RESTful web services using Spring Boot.

What is Web Services?

Web services are the types of internet software that uses standardized messaging protocol over the distributed environment. It integrates the web-based application using the **REST, SOAP, WSDL,** and **UDDI** over the network. For example, Java web service can communicate with .Net application.

Features of web Services

* Web services are designed for application to application interaction.
* It should be interoperable.
* It should allow communication over the network.

Components of Web Services

The web services must be able to fulfill the following conditions:

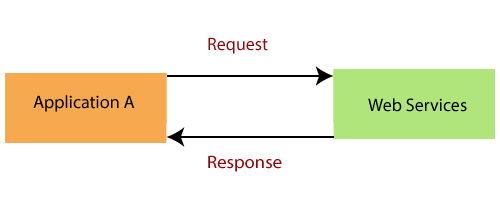
* The web service must be accessible over the internet.
* The web service is discoverable through a common mechanism like UDDI.
* It must be interoperable over any programming language or Operating System.

Uses of Web Services

* Web services are used for reusing the code and connecting the existing program.
* Web services can be used to link data between two different platforms.
* It provides interoperability between disparate applications.

How does data exchange between applications?

Suppose, we have an **Application A** which create a request to access the **web services**. The web services offer a list of services. The web service process the **request** and sends the **response** to the Application A. The input to a web service is called a request, and the output from a web service is called response. The web services can be called from different platforms.



There are two popular formats for request and response **XML** and **JSON**.

Internet interrupted: Amazon Web Services outage cripples airlines, banks, streaming services

**XML Format:** XML is the popular form as request and response in web services. Consider the following XML code:

1. <getDetail>
2. <id>DataStructureCourse</id>
3. </getDetail>

The code shows that user has requested to access the DataStrutureCourse. The other data exchange format is JSON. JSON is supported by wide variety of platform.

**JSON Format:** JSON is a readable format for structuring data. It is used for transiting data between server and web application.

1. [
2. "employee":
3. {
4. "id": 00987
5. "name":       "Jack",
6. "salary":      20000,
7. }
8. ]

To make a web service platform-independent, we make the **request** and **response** platform-independent.

Now a question arises, how does the **Application A** know the format of Request and Response?

The answer to this question is "Service Definition." Every web service offers a service definition. Service definition specifies the following:

* **Request/ Response format:** Defines the request format made by consumer and response format made by web service.
* **Request Structure:** Defines the structure of the request made by the application.
* **Response Structure:** Defines the structure of response returned by the web service.
* **Endpoint:** Defines where the services are available.

Key Terminology of Web Services

* Request and Response
* Message Exchange Format: XML and JSON
* Service Provider or Server
* Service Consumer or Client
* Service Definition
* Transport: HTTP and MQ

**Request and Response:** Request is the input to a web service, and the response is the output from a web service.

**Message Exchange Format:** It is the format of the request and response. There are two popular message exchange formats: **XML** and **JSON**.

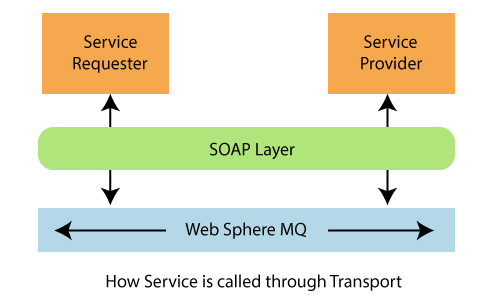
**Service Provider or Server:** Service provider is one which hosts the web service.

**Service Consumer or Client:** Service consumer is one who is using the web service.

Triggers in SQL (Hindi)

**Service Definition:** Service definition is the contract between the service provider and service consumer. Service definition defines the format of request and response, request structure, response structure, and endpoint.

**Transport:** Transport defines how a service is called. There is two popular way of calling a service: **HTTP** and Message Queue (**MQ**). By tying the URL of service, we can call the service over the internet. MQ communicates over the queue. The service requester puts the request in the queue. As soon as the service provider listens to the request. It takes the request, process the request, and create a response, and put the response back into MQ. The service requester gets the response from the queue. The communication happens over the queue.



# Characteristics of Web Services

Web services have the following characteristics:

* XML-based
* Coarse-grained
* Loosely coupled
* Capability to be synchronous and asynchronous
* Supports RPC

### **XML-based**

A web service uses XML at information representation and record transportation layer. Using XML, there is no need of networking, operating system, or platform binding. Web offering based application is highly interoperable application at their middle level.

### **Coarse-grained**

In the coarse-grained operation, a few objects hold a lot of related data. It provides broader functionality in comparison to fine-grained service. It wraps one or more fine-grained services together into a coarse-grained service. It is fine to have more coarse-grained service operations.

### **Loosely Coupled**

A web service supports **loosely coupled** connections between systems. It communicates by passing XML message to each other via a web API. Web API adds a layer of abstraction to the environment that makes the connection adaptable and flexible.

SQL CREATE TABLE

### **Capability to be synchronous and asynchronous**

Synchronous Web services are invoked over existing Web protocols by a client who waits for a response. Synchronous Web services are served by **RPC-oriented messaging**.

Asynchronous Web services are invoked over existing Web protocols by a client who does not wait for a response. The **document-oriented messaging** often used for asynchronous Web services. Asynchronous Web Service is a crucial factor in enabling loosely coupled system.

**Servlets, HTTP,** and **XML/SOAP** are used to implement synchronous or asynchronous endpoints.

### **Supports RPC**

A web service supports RPC through offering services of its personal, equivalent to those of a traditional aspect.

* A web service is a web resource. We can access a web service using platform-independent and language-neutral web protocols, such as HTTP. HTTP ensures easy integration of heterogeneous environment.
* A web service is typically registered. It can be located through a web service registry. A registry enables service consumers to find service that matches their needs. The service consumers may be human or other application.
* A web service provides an interface (a web API) that can be called from another program. The application-to-application programming can be invoked from any application.

Architecture of Web Services

The Web Services architecture describes how to instantiate the elements and implement the operations in an interoperable manner.

The architecture of web service interacts among three roles: **service provider, service requester,** and **service registry**. The interaction involves the three operations: **publish, find,** and **bind**. These operations and roles act upon the **web services artifacts**. The web service artifacts are the web service software module and its description.

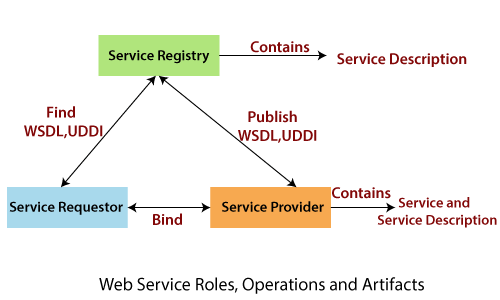
The service provider hosts a network-associable module (web service). It defines a service description for the web service and publishes it to a service requestor or service registry. These service requestor uses a find operation to retrieve the service description locally or from the service registry. It uses the service description to bind with the service provider and invoke with the web service implementation.

The following figure illustrates the operations, roles, and their interaction.

31.4M

682

C++ vs Java



Roles in a Web Service Architecture

There are three roles in web service architecture:

* Service Provider
* Service Requestor
* Service Registry

**Service Provider**

From an architectural perspective, it is the platform that hosts the services.

**Service Requestor**

Service requestor is the application that is looking for and invoking or initiating an interaction with a service. The browser plays the requester role, driven by a consumer or a program without a user interface.

**Service Registry**

Service requestors find service and obtain binding information for services during development.

Operations in a Web Service Architecture

Three behaviors that take place in the microservices:

* Publication of service descriptions **(Publish)**
* Finding of services descriptions **(Find)**
* Invoking of service based on service descriptions **(Bind)**

**Publish:** In the publish operation, a service description must be published so that a service requester can find the service.

**Find:** In the find operation, the service requestor retrieves the service description directly. It can be involved in two different lifecycle phases for the service requestor:

* At design, time to retrieve the service's interface description for program development.
* And, at the runtime to retrieve the service's binding and location description for invocation.

**Bind:** In the bind operation, the service requestor invokes or initiates an interaction with the service at runtime using the binding details in the service description to locate, contact, and invoke the service.

Artifacts of the web service

There are two artifacts of web services:

* Service
* Service Registry

**Service:** A service is an **interface** described by a service description. The service description is the implementation of the service. A service is a software module deployed on network-accessible platforms provided by the service provider. It interacts with a service requestor. Sometimes it also functions as a requestor, using other Web Services in its implementation.

**Service Description:** The service description comprises the details of the **interface** and **implementation** of the service. It includes its **data types, operations, binding information,** and **network location**. It can also categorize other metadata to enable discovery and utilize by service requestors. It can be published to a service requestor or a service registry.

Web Service Implementation Lifecycle

A web service implementation lifecycle refers to the phases for developing web services from the requirement to development. An Implementation lifecycle includes the following phases:

* Requirements Phase
* Analysis Phase
* Design Phase
* Coding Phase
* Test Phase
* Deployment Phase



**Requirements Phase**

The objective of the requirements phase is to understand the business requirement and translate them into the web services requirement. The requirement analyst should do requirement elicitation (it is the practice of researching and discovering the requirements of the system from the user, customer, and other stakeholders). The analyst should interpret, consolidate, and communicate these requirements to the development team. The requirements should be grouped in a centralized repository where they can be viewed, prioritized, and mined for interactive features.

**Analysis Phase**

The purpose of the analysis phase is to refine and translate the web service into conceptual models by which the technical development team can understand. It also defines the high-level structure and identifies the web service interface contracts.

**Design Phase**

In this phase, the detailed design of web services is done. The designers define web service interface contract that has been identified in the analysis phase. The defined web service interface contract identifies the elements and the corresponding data types as well as mode of interaction between web services and client.

**Coding Phase**

Coding and debugging phase is quite similar to other software component-based coding and debugging phase. The main difference lies in the creation of additional web service interface wrappers, generation of WSDL, and client stubs.

**Test Phase**

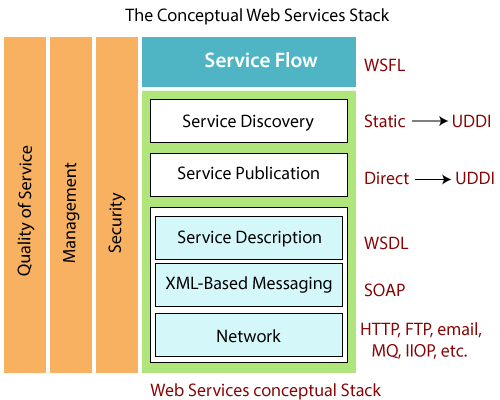
In this phase, the tester performs interoperability testing between the platform and the client's program. Testing to be conducted is to ensure that web services can bear the maximum load and stress. Other tasks like profiling of the web service application and inspection of the SOAP message should also perform in the test phase.

**Deployment Phase**

The purpose of the deployment phase is to ensure that the web service is properly deployed in the distributed system. It executes after the testing phase. The primary task of deployer is to ensure that the web service has been properly configured and managed. Other optional tasks like specifying and registering the web service with a UDDI registry also done in this phase.

Web Service Stack or Web Service Protocol Stack

To perform three operations: publish, find, and bind in an interoperable manner, there must be a **web service stack**. The web service stack embraces the standard at each level.



In the above figure, the top most layers build upon the capabilities provided by the lower layers. The three vertical towers represent the requirements that are applied at every level of the stack. The text on the right represents technologies that apply at that layer of the stack. A web service protocol stack typically stacks four protocols:

* Transport Protocol
* Messaging Protocol
* Description Protocol
* Discovery Protocol

**(Service) Transport Protocol:** The network layer is the foundation of the web service stack. It is responsible for transporting a message between network applications. HTTP is the network protocol for internet available web services. It also supports other network protocol such as **SMTP, FTP,** and **BEEP** (Block Extensible Exchange Protocol).

**(XML) Messaging Protocol:** It is responsible for encoding message in a common XML format so that they can understand at either end of a network connection. SOAP is the chosen XML messaging protocol because it supports three operations: publish, find, and bind operation.

**(Service) Description Protocol:** It is used for describing the public interface to a specific web service. WSDL is the standard for XML-based service description. WSDL describes the interface and mechanics of service interaction. The description is necessary to specify the **business context, quality of service,** and **service-to-service** relationship.

**(Service) Discovery Protocol:** It is a centralized service into a common registry so that network Web services can publish their location and description. It makes it easy to discover which services are available on the network.

The first three layers of the stack are required to provide or use any web service. The simplest stack consists of HTTP for the network layer, SOAP protocol for the XML-based messaging, and WSDL for the service description layer. These three-layer provides interoperability and enables web service to control the existing internet infrastructure. It creates a low cost of entry to a global environment.

The bottom three layers of the stack identify technologies for compliance and interoperability, the next two layer- **Service Publication** and **Service Discovery** can be implemented with a range of solutions.

# Types of Web Services

There are two types of web services:

* RESTful Web Servies
* SOAP Web Services

## RESTful Web Services

REST stands for **REpresentational State Transfer**. It is developed by **Roy Thomas Fielding** who also developed HTTP. The main goal of RESTful web services is to make web services **more effective**. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an **architectural approach**, not a protocol.

It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:

The resource has representations like XML, HTML, and JSON. The current state is captured by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

26.4M

529

Java Try Catch

* **GET:** It reads a resource.
* **PUT:** It updates an existing resource.
* **POST:** It creates a new resource.
* **DELETE:** It deletes the resource.

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

**POST /users:** It creates a user.

**GET /users/{id}:** It retrieve the detail of one user.

**GET /users:** It retrieve the detail of all users.

**DELETE /users:** It delete all users.

**DELETE /users/{id}:** It delete a user.

**GET /users/{id}/posts/post\_id:** It retrieve the detail of a specific post.

**POST / users/{id}/ posts:** It creates a post for a user.

**GET /users/{id}/post:** Retrieve all posts for a user

HTTP also defines the following standard status code:

* **404:** RESOURCE NOT FOUND
* **200:** SUCCESS
* **201:** CREATED
* **401:** UNAUTHORIZED
* **500:** SERVER ERROR

### **RESTful Service Constraints**

* There must be a service producer and service consumer.
* The service is stateless.
* The service result must be cacheable.
* The interface is uniform and exposing resources.
* The service should assume a layered architecture.

### **Advantages of RESTful web services**

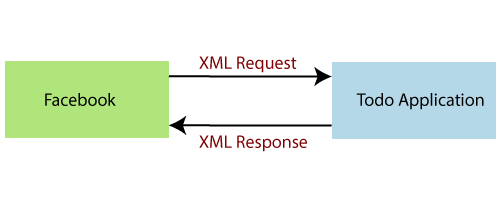
* RESTful web services are **platform-independent**.
* It can be written in any programming language and can be executed on any platform.
* It provides different data format like **JSON, text, HTML,** and **XML**.
* It is fast in comparison to SOAP because there is no strict specification like SOAP.
* These are **reusable**.
* These are **language neutral**.

## SOAP Web Services

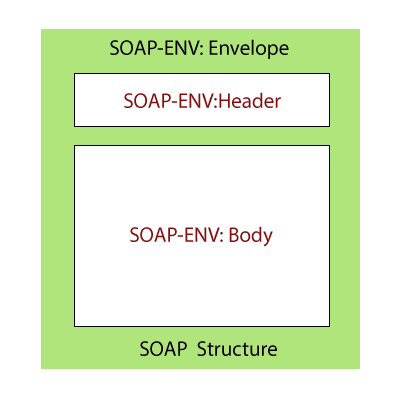
REST defines an architectural approach whereas SOAP poses a restriction on the format of the XML. XML transfer data between the service provider and service consumer. Remember that SOAP and REST are not **comparable**.

**SOAP:** SOAP acronym for **Simple Object Access Protocol**. It defines the standard XML format. It also defines the way of building web services. We use Web Service Definition Language (WSDL) to define the format of **request XML** and the **response XML**.

For example, we have requested to access the **Todo** application from the **Facebook** application. The Facebook application sends an XML request to the Todo application. Todo application processes the request and generates the XML response and sends back to the Facebook application.



If we are using SOAP web services, we have to use the **structure** of SOAP.



In the above figure, the **SOAP-Envelope** contains a **SOAP-Header** and **SOAP-Body**. It contains meta-information needed to identify the request, for example, authentication, authorization, signature, etc. SOAP-Header is optional. The **SOAP-Body** contains the real XML content of request or response. In case of an error, the response server responds back with SOAP-Fault.

Let's understand the SOAP XML request and response structure.

**XML Request**

1. <Envelop xmlns=?http://schemas.xmlsoap.org/soap/envelop/?>
2. <Body>
3. <getCourseDetailRequest xmlns=?http://udemy.com/course?>
4. <id>course1</id>
5. <getCourseDetailRequest>
6. </Body>
7. </Envelop>

**XML Response**

1. <SOAP-ENV:Envelope xmlns:SOAP-ENV=?http://schemas.xmlsoap.org/soap/envelope/?>
2. <SOAP-ENV:Header />             <!?empty header-->
3. <SOAP-ENV:Body>                <!?body begin-->
4. <ns2:getCourseDetailsResponse xmlns:ns2=?http://in28mi> <!--content of the response-->
5. <ns2:course>
6. <ns2:id>Course1</ns2:id>
7. <ns2:name>Spring<ns2:name>
8. <ns2:description>10 Steps</ns1:description>
9. </ns2:course>
10. </ns2:getCourseDetailResponse>
11. </SOAP-ENV:Body>       <!?body end-->
12. </SOAP-ENV:Envelope>

## Points to remember

* SOAP defines the format of **request** and **response**.
* SOAP does not pose any restriction on transport. We can either use **HTTP** or **MQ** for communication.
* In SOAP, service definition typically done using **Web Service Definition Language (WSDL)**. WSDL defines **Endpoint, All Operations, Request Structure,** and **Response Structure**.

The **Endpoint** is the connection point where HTML or ASP pages are exposed. It provides the information needed to address the Web Service endpoint. The operations are the services that are allowed to access. Request structure defines the structure of the request, and the response structure defines the structure of the response.

# Web Services Components

There are two components of web services:

* Web Service Description Language (WSDL)
* Universal Description Discovery and Integration (UDDI)

## Web Service Description Language (WSDL)

WSDL acronym for **Web Service Description Language**. WSDL is an XML based interface description language. It is used for describing the functionality offered by a web service. Sometimes it is also known as the WSDL file. The extension of the WSDL file is **.wsdl**. It provides the machine-readable description of how the service can be called, what parameter it expects, and what data structure it returns.

It describes service as a collection of network endpoint, or ports. It is often used in combination with SOAP and an XML schema to provide XML service over the distributed environment. In short, the purpose of WSDL is similar to type-signature in a programming language.

The current version of WSDL is 2.0. Version 1.1 does not endorse by W3C.

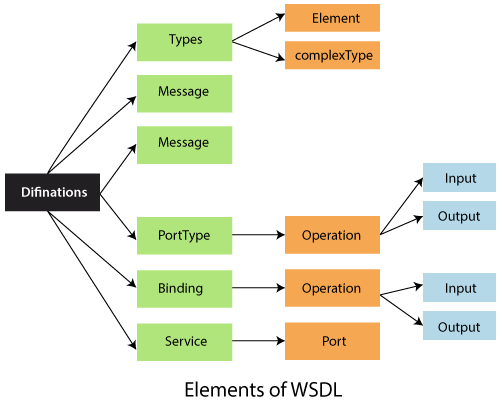
30M

694

OOPs Concepts in Java

### **Elements of WSDL**

|  |  |  |
| --- | --- | --- |
| **WSDL 1.1 Term** | **WSDL 2.0 Term** | **Description** |
| Service | Service | It is a set of system functions. |
| Port | Endpoint | It is an endpoint that defines a combination of binding and network addresses. |
| Binding | Binding | It specifies the interface and defines the SOAP binding style. It also defines the operations. |
| PortType | Interface | An abstract set of services supported by one or more endpoints. |
| Operation | Operation | Abstract detail of an action supported by the service. It defines the SOAP actions and the way of encoding the message. |
| Message | N/A | An abstract, typed definition of data to communicate. W3C has removed the message in WSDL 2.0, in which XML Schema types for defining bodies of inputs, outputs, and faults are referred directly. |
| Types | Types | It is a container for data type definition. The XML Schema language (XSD) is used for this purpose. |



## Universal Description, Discovery, and Integration (UDDI)

UDDI acronym for **Universal Description, Discovery,** and **Integration**. It is an **XML-based** registry for businesses word wide to list themselves on the internet. It defines a set of services supporting the description and discovery of the business, organizations, or other web service providers. The UDDI makes the services available and the technical interfaces which may be used to access those services.

The idea behind UDDI is to discover organizations and the services that organizations offer, much like using a telephone directory. It allows the business to list themselves by **name, product, location,** or the **web service** they offer. A UDDI works in the following manner:

* A service provider registers its business with the UDDI registry.
* A service provider registers each service separately with the UDDI registry.
* The consumer looks up the business and service in the UDDI registry.
* The consumer binds the service with the service provider and uses the service.

The UDDI business registry system has three directories are as follows:

* White Pages
* Yellow pages
* Green Pages

**White Pages:** The white pages contain basic information such as **company name, address, phone number,** and other business identifiers such as tax numbers.

**Yellow Pages:** The yellow pages contain detailed business data organized by relevant business classification. The version of the yellow page classifies business according to the newer **NAICS** (North American Industry Classification System).

**Green Pages:** The green pages contain information about the company's crucial business process, such as **operating platform, supported programs,** and **other high-level business protocols**.

# Difference between SOAP and RESTful Web Services

|  |  |
| --- | --- |
| **SOAP Protocol** | **RESTful Web Services** |
| SOAP is a protocol. | REST is an architectural approach. |
| SOAP acronym for Simple Object Access Protocol. | REST acronym for REpresentational State Transfer. |
| In SOAP, the data exchange format is always XML. | There is no strict data exchange format. We can use JSON, XML, etc. |
| XML is the most popular data exchange format in SOAP web services. | JSON is the most popular data exchange format in RESTful web services. |
| SOAP uses Web Service Definition Language (WSDL). | REST does not have any standard definition language. |
| SOAP does not pose any restrictions on transport. We can use either HTTP or MQ. | RESTful services use the most popular HTTP protocol. |
| SOAP web services are typical to implement. | RESTful services are easier to implement than SOAP. |
| SOAP web services use the JAX-WS API. | RESTful web services use the JAX-RS API. |
| SOAP protocol defines too many standards. | RESTful services do not emphasis on too many standards. |
| SOAP cannot use RESTful services because it is a protocol. | RESTful service can use SOAP web services because it is an architectural approach that can use any protocol like HTTP and SOAP. |
| SOAP reads cannot be cached. | REST reads can be cached. |

Introduction to RESTful Web Services

REST stands for **REpresentational State Transfer**. It is developed by **Roy Thomas Fielding**, who also developed HTTP. The main goal of RESTful web services is to make web services **more effective**. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an **architectural approach**, not a protocol.

It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:

The resource has representations like XML, HTML, and JSON. The current state capture by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

* **GET:** It reads a resource.
* **PUT:** It updates an existing resource.
* **POST:** It creates a new resource.
* **DELETE:** It deletes the resource.

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

Prime Ministers of India | List of Prime Minister of India (1947-2020)

**POST /users:** It creates a user.

**GET /users/{id}:** It retrieves the detail of a user.

**GET /users:** It retrieves the detail of all users.

**DELETE /users:** It deletes all users.

**DELETE /users/{id}:** It deletes a user.

**GET /users/{id}/posts/post\_id:** It retrieve the detail of a specific post.

**POST / users/{id}/ posts:** It creates a post of the user.

Further, we will implement these URI in our project.

HTTP also defines the following standard status code:

* **404:** RESOURCE NOT FOUND
* **200:** SUCCESS
* **201:** CREATED
* **401:** UNAUTHORIZED
* **500:** SERVER ERROR

RESTful Service Constraints

* There must be a service producer and service consumer.
* The service is stateless.
* The service result must be cacheable.
* The interface is uniform and exposing resources.
* The service should assume a layered architecture.

Advantages of RESTful web services

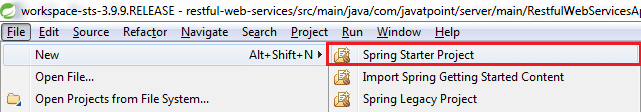
* RESTful web services are **platform-independent**.
* It can be written in any programming language and can be executed on any platform.
* It provides different data format like **JSON, text, HTML,** and **XML**.
* It is fast in comparison to SOAP because there is no strict specification like SOAP.
* These are **reusable**.
* They are **language neutral**.

# Initializing a RESTful Web Services Project with Spring Boot

**Step 1:** Download the **Spring Tool Suite (STS)** from <https://spring.io/tools3/sts/all> and extract it.

**Step 2:** Launch the **STS**.

**Step 3:** Click on **File menu -> New -> Spring Starter Project ->**

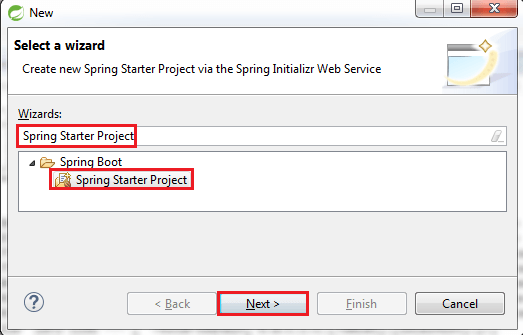


If the **Spring Starter Project** is not enlisted, then click on **Other** at the bottom of the menu. A dialog box appears on the screen. Type **Spring Starter Project** in the **Wizards** text box and click on the **Next** button.

25.4M

526

HTML Tutorial



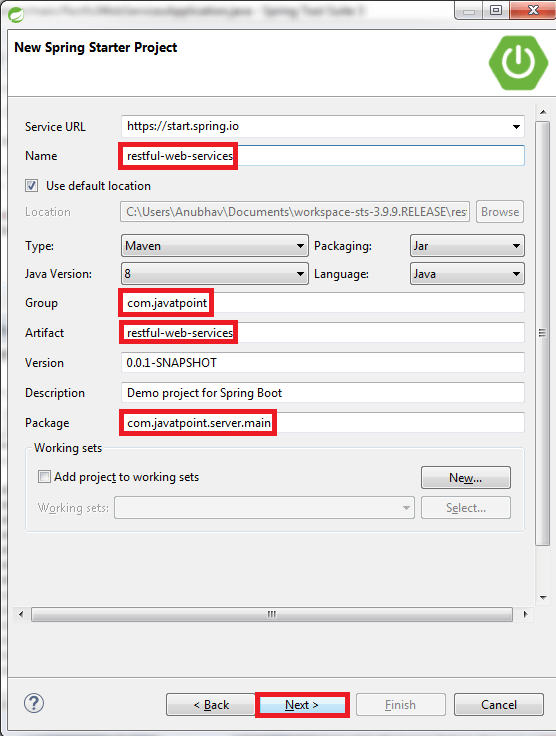
**Step 4:** provide the name, group, and package of the project. We have provided:

Name: **restful-web-services**

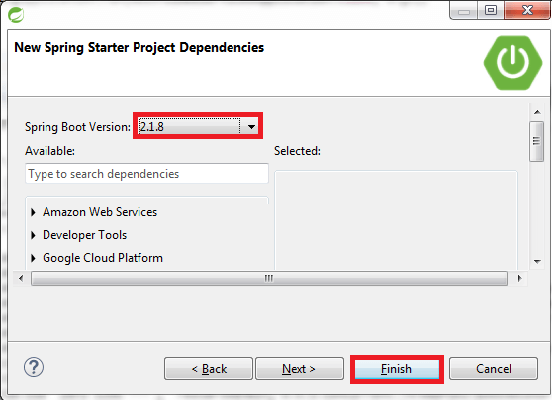
Group: **com.javatpoint**

Package: **com.javatpoint.server.main**

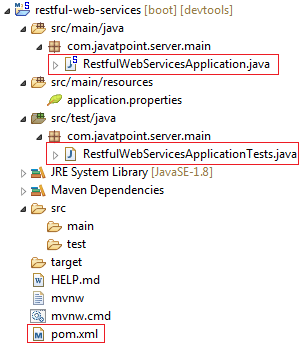
Click on the **Next** button.



**Step 5:** Choose the Spring Boot Version **2.1.8**.



**Step 6:** We can see the project structure in the project explorer window.



**Step 7:** Go to the Maven Repository <https://mvnrepository.com/> and add **Spring Web MVC, Spring Boot DevTools, JPA,** and **H2** dependencies in the pom.xml. After adding the dependencies, the pom.xml file looks like the following:

**pom.xml**

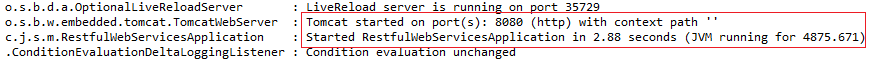
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
6. <parent>
7. <groupId>org.springframework.boot</groupId>
8. <artifactId>spring-boot-starter-parent</artifactId>
9. <version>2.1.8.RELEASE</version>
10. <relativePath/> <!-- lookup parent from repository -->
11. </parent>
12. <groupId>com.javatpoint</groupId>
13. <artifactId>restful-web-services</artifactId>
14. <version>0.0.1-SNAPSHOT</version>
15. <name>restful-web-services</name>
16. <description>Demo project **for** Spring Boot</description>
17. <properties>
18. <java.version>1.8</java.version>
19. </properties>
20. <dependencies>
21. <dependency>
22. <groupId>org.springframework.boot</groupId>
23. <artifactId>spring-boot-starter</artifactId>
24. </dependency>
25. <dependency>
26. <groupId>org.springframework.boot</groupId>
27. <artifactId>spring-boot-starter-activemq</artifactId>
28. </dependency>
29. <dependency>
30. <groupId>org.springframework.boot</groupId>
31. <artifactId>spring-boot-starter-web</artifactId>
32. </dependency>
33. <dependency>
34. <groupId>org.springframework.boot</groupId>
35. <artifactId>spring-boot-starter-tomcat</artifactId>
36. </dependency>
37. <dependency>
38. <groupId>org.springframework</groupId>
39. <artifactId>spring-webmvc</artifactId>
40. </dependency>
41. <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-devtools -->
42. <dependency>
43. <groupId>org.springframework.boot</groupId>
44. <artifactId>spring-boot-devtools</artifactId>
45. <scope>runtime</scope>
46. </dependency>
47. <!-- https://mvnrepository.com/artifact/org.hibernate.javax.persistence/hibernate-jpa-2.1-api -->
48. <dependency>
49. <groupId>org.hibernate.javax.persistence</groupId>
50. <artifactId>hibernate-jpa-2.1-api</artifactId>
51. <version>1.0.0.Final</version>
52. </dependency>
53. <!-- https://mvnrepository.com/artifact/com.h2database/h2 -->
54. <dependency>
55. <groupId>com.h2database</groupId>
56. <artifactId>h2</artifactId>
57. <scope>runtime</scope>
58. </dependency>
59. <dependency>
60. <groupId>org.apache.maven</groupId>
61. <artifactId>maven-archiver</artifactId>
62. <version>2.5</version>
63. </dependency>
64. <dependency>
65. <groupId>org.springframework.boot</groupId>
66. <artifactId>spring-boot-starter-test</artifactId>
67. <scope>test</scope>
68. </dependency>
69. </dependencies>
70. <build>
71. <plugins>
72. <plugin>
73. <groupId>org.springframework.boot</groupId>
74. <artifactId>spring-boot-maven-plugin</artifactId>
75. </plugin>
76. </plugins>
77. </build>
78. </project>

**Step 8:** Now open the **RestfulWebServicesApplication.java** file and Run the file as Java Application.

1. **package** com.javatpoint.server.main;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** RestfulWebServicesApplication
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(RestfulWebServicesApplication.**class**, args);
10. }
11. }

It does not perform any service but ensures that the application is running properly.

**Output**



## Creating a Hello World Service

**Step 1:** Create a new class with the name **HelloWorldController** in the package **com.javatpoint.server.main**.

**Step 2:** Whenever we create a web service, we need to define two things **Get** method and the **URI**. Now create the **helloWorld()** method which returns the string "Hello World." If we want to tell the spring MVC that it is going to handle the REST request, we have to add **@RestController** annotation. Now it becomes a rest controller which can handle the Rest request.

The next thing we have to do is create a mapping for the method. Add **@RequestMapping** annotation just above the helloWorld() method. The HelloWorldController looks like the following:

1. **package** com.javatpoint.server.main;
2. **import** org.springframework.web.bind.annotation.RequestMapping;
3. **import** org.springframework.web.bind.annotation.RequestMethod;
4. **import** org.springframework.web.bind.annotation.RestController;
5. //Controller
6. @RestController
7. **public** **class** HelloWorldController
8. {
9. //using get method and hello-world as URI
10. @RequestMapping(method=RequestMethod.GET, path="/hello-world")
11. **public** String helloWorld()
12. {
13. **return** "Hello World";
14. }
15. }

We can also improve the above code by using the **@GetMapping** annotation instead of @RequestMapping. Here the method specification is not required.

1. **package** com.javatpoint.server.main;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. //Controller
5. @RestController
6. **public** **class** HelloWorldController
7. {
8. //using get method and hello-world as URI
9. @GetMapping(path="/hello-world")
10. **public** String helloWorld()
11. {
12. **return** "Hello World";
13. }
14. }

**Step 3:** Run the **RestfulWebServiceApplication**. It displays the string **Hello World** on the browser.

## Enhancing the Hello World Service to Return a Bean

In this section, we are going to generate a bean for the method helloWorld().

**Step 1:** Create a **helloWorldBean()** method in **HelloWordController.java** file. Map the URI to "**/hello-world-bean**" and return **HelloWorldBean**.

**HelloWorldController.java**

1. **package** com.javatpoint.server.main;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. //Controller
5. @RestController
6. **public** **class** HelloWorldController
7. {
8. //using get method and hello-world URI
9. @GetMapping(path="/hello-world")
10. **public** String helloWorld()
11. {
12. **return** "Hello World";
13. }
14. @GetMapping(path="/hello-world-bean")
15. **public** HelloWorldBean helloWorldBean()
16. {
17. **return** **new** HelloWorldBean("Hello World"); //constructor of HelloWorldBean
18. }
19. }

**Step 2:** Create a class **HelloWorldBean**.

**Step 3:** Generate Getters and **Setters**.

Right-click -> Source -> Generate Getters and Setters -> check the box -> Ok

**Step 4:** Generate **toString()**..

Right-click -> Source -> Generate toString().. -> Ok

**HelloWorldBean.java**

1. **package** com.javatpoint.server.main;
2. **public** **class** HelloWorldBean
3. {
4. **public** String message;
5. //constructor of HelloWorldBean
6. **public** HelloWorldBean(String message)
7. {
8. **this**.message=message;
9. }
10. //generating getters and setters
11. **public** String getMessage()
12. {
13. **return** message;
14. }
15. **public** **void** setMessage(String message)
16. {
17. **this**.message = message;
18. }
19. @Override
20. //generate toString
21. **public** String toString()
22. {
23. **return** String.format ("HelloWorldBean [message=%s]", message);
24. }
25. }

**Step 5:** Launch the **HelloWorldController**. The URL of the browser changes to **localhost:8080/hello-world-bean**.

It returns the message "**Hello World**" in JSON format.

1. {
2. message: "Hello World"
3. }

Spring Boot Auto Configuration and Dispatcher Servlet

In this section, we will see the background functionality of the application:

What is dispatcher servlet?

Who is configuring dispatcher servlet?

What does dispatcher servlet do?

32.2M

739

Features of Java - Javatpoint

How does the HelloWorldBean object gets converted to JSON?

Who is configuring the error mapping?

Spring Boot Auto Configuration

* Spring Boot automatically configures a spring application based on dependencies present or not present in the classpath as a jar, beans, properties, etc.
* It makes development easier and faster as there is no need to define certain beans that are included in the auto-configuration classes.
* A typical MVC database driven Spring MVC application requires a lot of configuration such as **dispatcher servlet, a view resolver, Jackson, data source, transaction manager,** among many others.
  + Spring Boot auto-configures a **Dispatcher Servlet** if **Spring MVC jar** is on the classpath.
  + Auto-configures the **Jackson** if **Jackson jar** is on the classpath.
  + Auto-configures a **Data Source** if **Hibernate jar** is on the classpath.
* Auto-configuration can be enabled by adding **@SpringBootApplication** or **@EnableAutoConfiguration** annotation in startup class. It indicates that it is a spring context file.
* It enables something called **auto-configuration**.
* It enable something called **components scan**. It is the features of Spring where it will start automatically scanning classes in the package and sub package for any bean file.
* There is some example of auto configuration done by Spring Boot:
  + **DispatcherServletAutoConfiguration**
  + **DataSourceAutoConfiguration**
  + **JacksonAutoConfiguration**
  + **ErrorMvcAutoConfiguration** (#basicErrorController)
* We can see the auto-configuration done by Spring Boot in the **AUTO-CONFIGURATION REPORT** or **CONDITIONS EVALUATION REPORT**.
* Classes can be **excluded** from auto-configuration by adding:

1. @SpringBootApplication (exclude={JacksonAutoConfiguration.**class**, JmxAutoConfiguration.**class**})

Or add the following statement in the **application.properties** file.

1. spring.autoconfiguration.exclude=org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration

We exclude classes form the auto-configuration for **faster startup** and **better performance** of the application.

* AUTO-CONFIGURATION REPORT generated by enabling **debug** mode. Open the **application.properties** file and add the following statement:

1. logging.level.org.springframework=debug

Run RestfulWebServiceApplication.java file. It shows the Positive matches, Negative matches, Exclusions, and Unconditional classes under the CONDITIONS EVALUATION REPORT.

1. -----------------------------------------------
2. CONDITIONS EVALUATION REPORT
3. -----------------------------------------------
4. Positive matches:
5. ----------------------
6. ActiveMQAutoConfiguration matched:
7. - @ConditionalOnClass found required classes 'javax.jms.ConnectionFactory', 'org.apache.activemq.ActiveMQConnectionFactory' (OnClassCondition)
8. - @ConditionalOnMissingBean (types: javax.jms.ConnectionFactory; SearchStrategy: all) did not find any beans (OnBeanCondition)
9. ------------------
10. ------------------
11. ------------------
12. Negative matches:
13. -------------------------
14. ActiveMQConnectionFactoryConfiguration.PooledConnectionFactoryConfiguration: Did not match:
15. -@ConditionalOnClass did not find required classes 'org.messaginghub.pooled.jms.JmsPoolConnectionFactory','org.apache.commons.pool2.PooledObject' (OnClassCondition)
16. ----------------
17. ----------------
18. ----------------
19. Exclusions:
20. ---------------
21. None
22. Unconditional classes:
23. -----------------------------    org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration
24. org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration

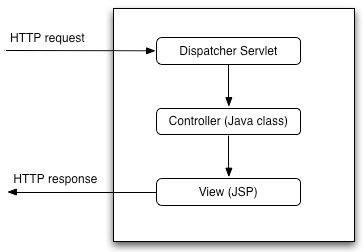
There is a lot of information inside the report, so it is not possible to show all the detail. If we scroll down the report and have a closure look at auto configuration log, we found **DispatcherServletAutoConfiguration matched**.

1. DispatcherServletAutoConfiguration matched:
2. -@ConditionalOnClass found required **class** 'org.springframework.web.servlet.DispatcherServlet' (OnClassCondition)
3. - found 'session' scope (OnWebApplicationCondition)

It is because, the dependency **spring-boot-starter-web** depends on **spring-webmvc** dependency. Therefore we get the DispatcherServlet class in our classpath.

Dispatcher Servlet

In Spring MVC all incoming requests go through a single servlet is called **Dispatcher Servlet (front controller)**. The front controller is a design pattern in web application development. A single servlet receives all the request and transfers them to all other components of the application.



The job of DispatcherServlet is to take an incoming URI and find the right combination of **handlers** (Controller classes) and **views** (usually JSPs). When the DispatcherServlet determines the view, it renders it as the response. Finally, the DispatcherServlet returns the Response Object to back to the client. In short, the Dispatcher Servlet plays the key role.

The other thing to notice is that ErrorMvcAutoConfiguration:

1. ErrorMvcAutoConfiguration matched:
2. -@ConditionalOnClass found required classes 'javax.servlet.Servlet', 'org.springframework.web.servlet.DispatcherServlet' (OnClassCondition)- found 'session' scope (OnWebApplicatiossssnCondition)

It configures the **basicErrorController, errorAttributes, ErrorMvcAutoConfiguration,** and **DefaultErrorViewResolverConfiguration**. It creates the default error page which is known as **Whitelabel Error Page**.



The other thing which is auto-configured **HttpMessageConvertersAutoConfiguration**. These message converter automatically converts the message.

1. HttpMessageConvertersAutoConfiguration matched:
2. -@ConditionalOnClass found required **class** 'org.springframework.http.converter.HttpMessageConverter' (OnClassCondition)
3. ----------------
4. -----------------
5. JacksonAutoConfiguration.Jackson2ObjectMapperBuilderCustomizerConfiguration matched: - @ConditionalOnClass found required **class** 'org.springframework.http.converter.json.Jackson2ObjectMapperBuilder'(OnClassCondition)

It initializes the Jackson bean and the message converter. The **Jackson2ObjectMapper** does the conversion from **bean to JSON** and **JSON to bean**.

Enhancing the Hello World Service with a Path Variable

The **@PathVariable** annotation is used to extract the value from the URI. It is most suitable for the RESTful web service where the URL contains some value. Spring MVC allows us to use multiple @PathVariable annotations in the same method. A path variable is a critical part of creating rest resources.

We will create another hello-world-bean request with a path parameter.

**Step 1:** Open the **HelloWorldController.java** file and add another helloWorldBean() service.

**HelloWorldController.java**

32.2M

739

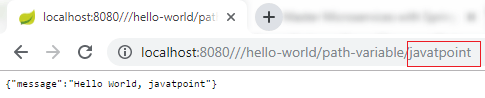
Features of Java - Javatpoint

1. **package** com.javatpoint.server.main;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.PathVariable;
4. **import** org.springframework.web.bind.annotation.RestController;
5. **import** org.springframework.context.annotation.Configuration;
6. @Configuration
7. //Controller
8. @RestController
9. **public** **class** HelloWorldController
10. {
11. //using get method and hello-world URI
12. @GetMapping(path="/hello-world")
13. **public** String helloWorld()
14. {
15. **return** "Hello World";
16. }
17. @GetMapping(path="/hello-world-bean")
18. //method- which returns "Hello World"
19. **public** HelloWorldBean helloWorldBean()
20. {
21. **return** **new** HelloWorldBean("Hello World");//constructor of HelloWorldBean
22. }
23. //passing a path variable
24. @GetMapping(path="/hello-world/path-variable/{name}")
25. **public** HelloWorldBean helloWorldPathVariable(@PathVariable String name)
26. {
27. **return** **new** HelloWorldBean(String.format("Hello World, %s", name)); //%s replace the name
28. }
29. }

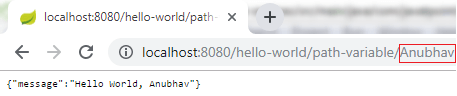
Whatever value we will pass to the path variable is picked up by the controller and returned to the response.

**Step 2:** Type the URL http://localhost:8080///hello-world/path-variable/javatpoint

**Step 3:** Run the **HelloWorldController.java** file. We get the following response on the browser.



Let's change the path variable once again: http://localhost:8080/hello-world/path-variable/Anubhav



We can see that whatever we are writing in the path variable is returned to the response.

Creating User Bean and User Service

In this section, we are going to create real resources users and the post. We will use a static array list to represent the data.

**Step 1:** Create a new package with the name **com.javatpoint.server.main.user**.

**Step 2:** Create a bean class (User) to store the user detail.

Right click on the package **user -> New -> Class -> Provide Name ->** Finish. Here, we have provided class name User.

**Step 3:** Define three private variables **id, name,** and **dob**.

**Step 4:** Generate **Getters** and **Setters**.

Right click on **the file -> Source -> Generate Getters and Setters... ->Select All -> Generate**.

**Step 5:** Generate **toString**.

Right click on **the file -> Source -> Generate toString... ->Select All -> Generate**.

**Step 6:** Generate **Constructors**.

Right click on **the file -> Source -> Generate Constructor using Fields... -> Generate**.

**User.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.Date;
3. **public** **class** User
4. {
5. **public** User(Integer id, String name, Date dob)
6. {
7. **super**();
8. **this**.id = id;
9. **this**.name = name;
10. **this**.dob = dob;
11. }
12. **private** Integer id;
13. **private** String name;
14. **private** Date dob;
15. **public** Integer getId()
16. {
17. **return** id;
18. }
19. **public** **void** setId(Integer id)
20. {
21. **this**.id = id;
22. }
23. **public** String getName()
24. {
25. **return** name;
26. }
27. **public** **void** setName(String name)
28. {
29. **this**.name = name;
30. }
31. **public** Date getDob()
32. {
33. **return** dob;
34. }
35. **public** **void** setDob(Date dob)
36. {
37. **this**.dob = dob;
38. }
39. @Override
40. **public** String toString()
41. {
42. //return "User [id=" + id + ", name=" + name + ", dob=" + dob + "]";
43. **return** String.format("User [id=%s, name=%s, dob=%s]", id, name, dob);
44. }
45. }

Before moving to the next step first move **HelloWorldBean.java** and **HelloWorldController.java** in the package **com.javatpoint.server.main.helloworld**.

**Step 7:** Create a class with name **UserDaoService** in the package **com.javatpoint.server.main.user**.

**UserDaoService.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.ArrayList;
3. **import** java.util.Date;
4. **import** java.util.List;
5. **import** org.springframework.stereotype.Component;
6. @Component
7. **public** **class** UserDaoService
8. {
9. **public** **static** **int** usersCount=5;
10. //creating an instance of ArrayList
11. **private** **static** List<User> users=**new** ArrayList<>();
12. //static block
13. **static**
14. {
15. //adding users to the list
16. users.add(**new** User(1, "John", **new** Date()));
17. users.add(**new** User(2, "Robert", **new** Date()));
18. users.add(**new** User(3, "Adam", **new** Date()));
19. users.add(**new** User(4, "Andrew", **new** Date()));
20. users.add(**new** User(5, "Jack", **new** Date()));
21. }
22. //method that retrieve all users from the list
23. **public** List<User> findAll()
24. {
25. **return** users;
26. }
27. //method that add the user in the list
28. **public** User save(User user)
29. {
30. **if**(user.getId()==**null**)
31. {
32. //increments the user id
33. user.setId(++usersCount);
34. }
35. users.add(user);
36. **return** user;
37. }
38. //method that find a particular user from the list
39. **public** User findOne(**int** id)
40. {
41. **for**(User user:users)
42. {
43. **if**(user.getId()==id)
44. **return** user;
45. }
46. **return** **null**;
47. }
48. }

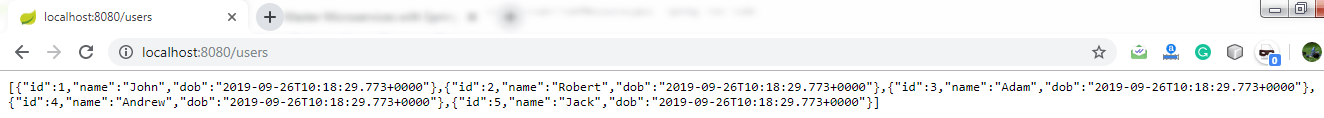
Implementing Get Methods for User Resource

**Step 8:** Now create a user controller class with name **UserResource**.

**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.RestController;
6. @RestController
7. **public** **class** UserResource
8. {
9. @Autowired
10. **private** UserDaoService service;
11. @GetMapping("/users")
12. **public** List<User> retriveAllUsers()
13. {
14. **return** service.findAll();
15. }
16. }

**Step 9:** Run the application and type the **localhost:8080/users** in the address bar of the browser. It returns the users list in **JSON** format.



If the date is displaying in the default timestamp format as:

**"dob": "1500370250075"**

We need to do setting for proper date format.

Open the **application.properties** file. Remove the debug configuration and add the following configuration:

1. spring.jackson.serialization.write-dates-as-timestamps=**false**

The above statement telling the Jackson framework that when serializing don't treat the date as a timestamp.

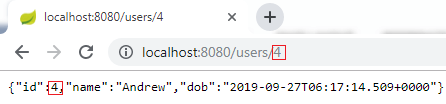
**Step 10:** If we want to display a specific user detail on the browser, add the mapping "**/user/{id}**" and create a method **retriveUser()** in the UserResource.

**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.PathVariable;
6. **import** org.springframework.web.bind.annotation.RestController;
7. @RestController
8. **public** **class** UserResource
9. {
10. @Autowired
11. **private** UserDaoService service;
12. @GetMapping("/users")
13. **public** List<User> retriveAllUsers()
14. {
15. **return** service.findAll();
16. }
17. //retrieves a specific user detail
18. @GetMapping("/users/{id}")
19. **public** User retriveUser(@PathVariable **int** id)
20. {
21. **return** service.findOne(id);
22. }
23. }

**Step 11:** Run the application and type **localhost:8080/users/{id}** in the browser. It returns the detail of the specific user id which we are passing in the path variable.

In the following image, we have retrieved the detail of the user having id **4**.



# Implementing the POST Method to create User Resource

In the previous few steps, we have created simple RESTful services. In this section, we will use the POST method to post the user resource for the specific URI "**/users**."

Here we are using two annotations, **@RequestBody** and **@PathMapping**.

## @RequestBody

The @RequestBody annotation maps body of the web request to the method parameter. The body of the request is passed through an HttpMessageConverter. It resolves the method argument depending on the content type of the request. Optionally, automatic validation can be applied by annotating the argument with @Valid.

In the following example, when we pass the **@RequestBody** annotation in the **createUser()** method, it maps to the **user** parameter.

10 Sec

31.4M

682

C++ vs Java

## @PathMapping

The @PathMapping annotation is the specialized version of the **@RequestMapping** annotation which acts as a shortcut for **@RequestMapping(method=RequestMethod=POST)**. @PostMapping method handles the Http POST requests matched with the specified URI.

Let's create a user resource and post that resource through the POST method.

**Step 1:** Open UserResource.java and add **@PostMapping("/user")**.

**Step 2:** Create a method **createUser()** and pass the User class's **object** as the **body** of the web.

**Step 3:** Save the created user.

**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.PathVariable;
6. **import** org.springframework.web.bind.annotation.PostMapping;
7. **import** org.springframework.web.bind.annotation.RequestBody;
8. **import** org.springframework.web.bind.annotation.RestController;
9. @RestController
10. **public** **class** UserResource
11. {
12. @Autowired
13. **private** UserDaoService service;
14. @GetMapping("/users")
15. **public** List<User> retriveAllUsers()
16. {
17. **return** service.findAll();
18. }
19. //retrieves a specific user detail
20. @GetMapping("/users/{id}")
21. **public** User retriveUser(@PathVariable **int** id)
22. {
23. **return** service.findOne(id);
24. }
25. //method that posts a new user detail
26. @PostMapping("/users")
27. **public** **void** createUser(@RequestBody User user)
28. {
29. User sevedUser=service.save(user);
30. }
31. }

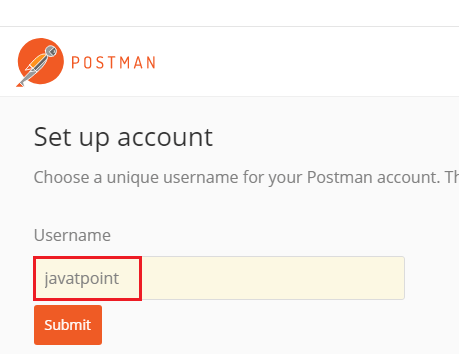
When we refresh the page it returns the get request. But we need to send POST request. We can send POST request through **REST client**. REST client is a client that is designed to use a service (RESTful) from a server.

Lets' see how to use the REST client.

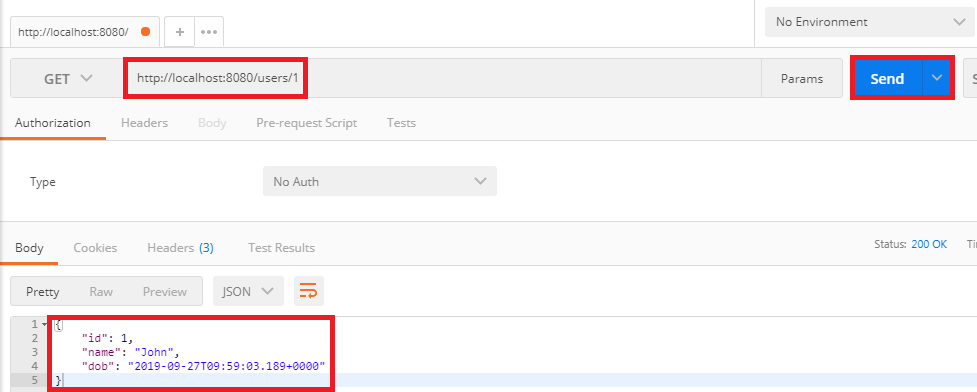
**Step 4:** Download the Postman from <https://www.getpostman.com/downloads/>.

Or add Google Chrome extension in the browser <https://bit.ly/1HCOCwF>.

Step 5: Launch the **Postman** and **Signup**. Create a **user name**. Here we have created user name **javatpoint** and clicked on **Submit** button. Consider the below image:

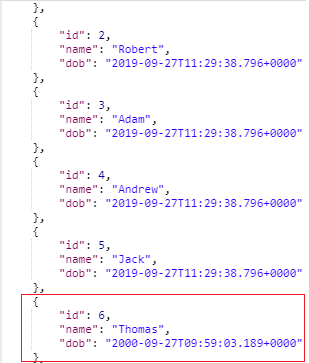


**Step 6:** First, we check for the **GET** request. Type the URL http://localhost:8080/users/1 in the address bar and click on the **Send** button. It returns the detail of the first user.



**Step 7:** Now we send a POST request.

* Change the method to the **POST**.
* Copy the **body** of the response coming from "/users/1".
* Click on **Body** tab. Now we create a body for the POST request.
* Choose the **raw** option. It creates a raw request.
* Paste the copied content.
* Remove the **id** because it increments automatically.
* Change the "**name**": "**Thomas**".
* Instead of Text we are sending the data in the JSON format. So select **JSON (application/json)**.
* Type the URL http://localhost:8080/users and click on the **Send** button.
* Click on the **Get** request on the left side of the window.
* Now we will send a Get request again, so change the URL http://localhost:8080/users and click on the **Send** button. It displays all the users, including which we have created.



## Enhancing POST Method to Return Correct HTTP Code and Status Location

In this section, we will return the status (Created) and URI ("/users/6") of the user recourse which we have created.

### **ResponseEntity Class**

The **ResponseEntity** is a class which extends **HttpEntity** and HttpStatus class. It is defined in **org.springframework.http.RequestEntity**.

* It is used in **RestTemplate** and **@Controller** methods.
* It is used as parameter in **getForEntity()** and **exchange()** method.
* It is also used in Spring MVC, as a parameter in a @Controller method.

### **created() method of RequestEntity Class**

The **created()** method is the static method of **RequestEntity** class. It creates a **new builder** with a CREATED status and a location header set to the given URI.

**Syntax**

1. **public** **static** ResponseEntity.BodyBuilder created(URI location)

**Parameter:** It accepts the **location URI** as a parameter.

**Returns:** It returns the **created builder**.

All Http status codes are **Enum constant**, which is defined in the **HttpStatus** class.

### **ServletUriComponentsBuilder**

The **ServletUriComponentsBuilder** is a class which is defined in org.springframework.web.servlet.support.ServletUriComponentsBuilder. It extends **UriComponentsBuilder** class. It has additional static factory methods to create a link based on the current HttpServletRequest.

### **fromCurrentRequest() method**

It is similar to **fromRequest(HttpServletRequest)** method except the request is obtained through RequestContextHolder.

### **path() method**

The path() is the method of **UriComponentsBuilder** class. It appends the given path to the existing path of this builder. The given path may contain URI template variable.

**Syntax**

1. **public** UriBuilderBuilder path(String path)

**Parameter:** It accepts a **path** as a parameter.

**Returns:** It returns the **UriComponentsBuilder**.

### **buildAndExpand() method**

It builds UriComponents instance and replaces URI template variables with the values obtained from an array. It is the shortcut method which combines calls to build() and then UriComponents.expand(Object... uriVariableValues).

**Syntax**

1. **public** UriComponents buildAndExpand(Object...uriVariableValues)

**Parameter:** It accepts the **URI variable values** as a parameter.

**Returns:** It returns the **URI components** with extended values.

### **build() method**

It builds UriComponents instance from the various components contained in the builder.

**Syntax**

1. **public** UriComponents build()

**Parameter:** It does not accept any parameter.

**Returns:** It returns the **Uri Components**.

Let's see how to return the status of the created resource and how to set URI of the created resource in the response.

**Step 1:** Create a method that creates a user resource and returns the **ResponseEntity**.

**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.net.URI;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.http.ResponseEntity;
6. **import** org.springframework.web.bind.annotation.GetMapping;
7. **import** org.springframework.web.bind.annotation.PathVariable;
8. **import** org.springframework.web.bind.annotation.PostMapping;
9. **import** org.springframework.web.bind.annotation.RequestBody;
10. **import** org.springframework.web.bind.annotation.RestController;
11. **import** org.springframework.web.servlet.support.ServletUriComponentsBuilder;
12. @RestController
13. **public** **class** UserResource
14. {
15. @Autowired
16. **private** UserDaoService service;
17. @GetMapping("/users")
18. **public** List<User> retriveAllUsers()
19. {
20. **return** service.findAll();
21. }
22. //retrieves a specific user detail
23. @GetMapping("/users/{id}")
24. **public** User retriveUser(@PathVariable **int** id)
25. {
26. **return** service.findOne(id);
27. }
28. //method that posts a new user detail and returns the status of HTTP and location of the user resource
29. @PostMapping("/users")
30. **public** ResponseEntity<Object> createUser(@RequestBody User user)
31. {
32. User sevedUser=service.save(user);
33. URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(sevedUser.getId()).toUri();
34. **return** ResponseEntity.created(location).build();
35. }
36. }

**Step 2:** Now open REST client **Postman** and create a **POST** request.

**Step 3:** Click on the POST request under the **History** tab.

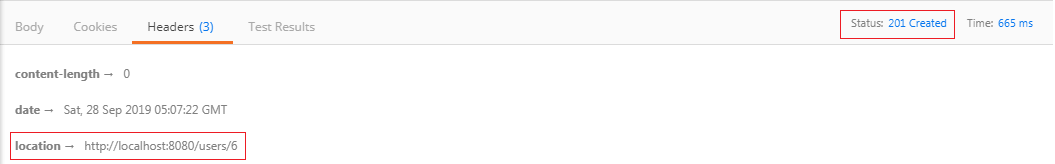
**Step 4:** Click on the Body tab and change the user name to **James**.

**Step 5:** Ensure that you have selected **JSON (application/json)** media type.

**Step 5:** Click on the **Send** button.

On the right-hand side of the window, we can see the **Status: 201 Created**. It means resource has been properly created.

**Step 6:** Now click on the Headers tab to see the location. Location is the URI of the created resource. It shows the location of the created user James that is "**/users/6**".



If the client wants to know where the user resources was created, just pick up the location from the header of the response.

Implementing Exception Handling- 404 Resource Not Found

In the previous section, we had returned the proper response status of CREATED when we created the resource. In this section, we will discuss what should be the response when a user resource does not exist.

Let's try and execute a simple response.

**Step 1:** Open Rest client **Postman** and select the **Get** method.

**Step 2:** Click on the **History** tab and choose the **Get** request.

10 Sec

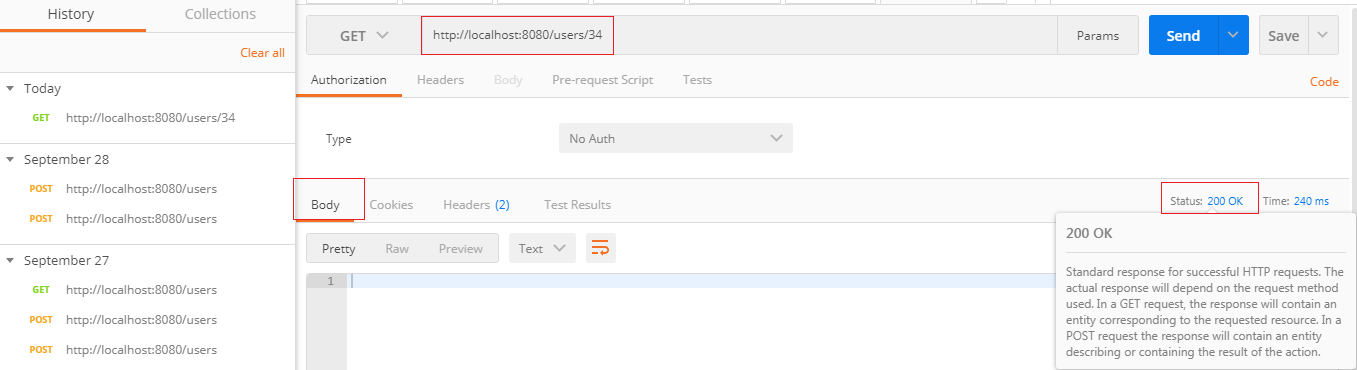
32.5M

627

Difference between JDK, JRE, and JVM

**Step 3:** Type the URI http://localhost:8080/users/{id}. The user id should not exist.

**Step 4:** Click on the **Send** Button.



We get the **Status: 200 OK** and **empty body** which is a successful response even though the resource does not exist. But it is not the proper response when a resource does not exist.

Let's fix that first.

**Step 1:** Open the **UserResource.java** file.

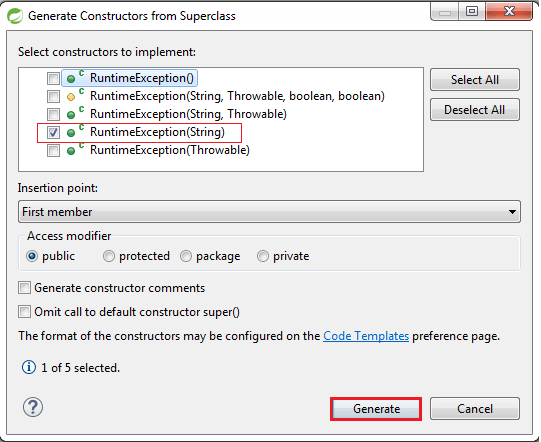
**Step 2:** Create a **UserNotFoundException**. It is a checked exception.

1. @GetMapping("/users/{id}")
2. **public** User retriveUser(@PathVariable **int** id)
3. {
4. User user= service.findOne(id);
5. **if**(user==**null**)
6. //runtime exception
7. **throw** **new** UserNotFoundException("id: "+ id);
8. **return** user;
9. }

**Step 3:** Create **UserNotFoundException** class.

**Step 4:** Generate Constructors from Superclass.

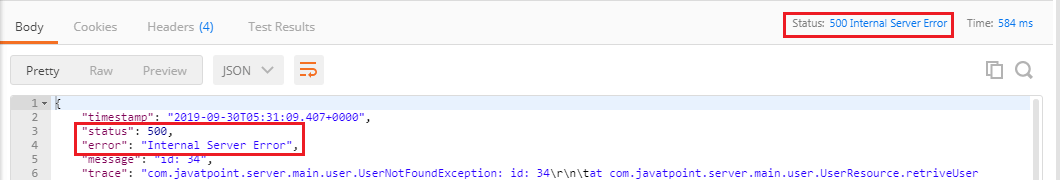
Right-click on the file -> Source -> Generate Constructors from Superclass... -> check the RuntimeException(String) -> Generate.



**UserNotFoundException.java**

1. **package** com.javatpoint.server.main.user;
2. **public** **class** UserNotFoundException **extends** RuntimeException
3. {
4. **public** UserNotFoundException(String message)
5. {
6. **super**(message);
7. }
8. }

**Step 5:** Open the Rest Client **Postman** and generate a Get response as we have done before. It shows the **Status: 500 Internal Server Error**.

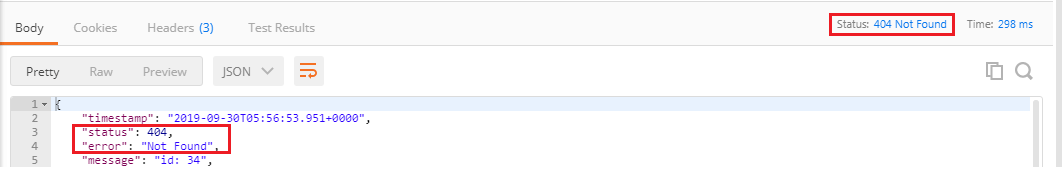


But the Status: 500 Internal Server Error is not the appropriate response for the resource not found. So, we will add an annotation **@ResponseStatus** to generate the Status: 404 Not Found.

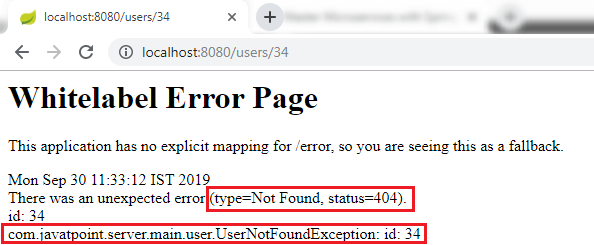
**UserNotFoundException.java**

1. **package** com.javatpoint.server.main.user;
2. **import** org.springframework.http.HttpStatus;
3. **import** org.springframework.web.bind.annotation.ResponseStatus;
4. @ResponseStatus(HttpStatus.NOT\_FOUND)
5. **public** **class** UserNotFoundException **extends** RuntimeException
6. {
7. **public** UserNotFoundException(String message)
8. {
9. **super**(message);
10. }
11. }

**Step 6:** Again move to **Postman** and generate a **Get** request.



We get the proper response **Status: 404 Not Found** when a user resource does not exist. The body of the request provided by default error handling that's why we are getting this return status back.



The combination of Spring Boot and Spring Web MVC framework provides error handling. Spring Boot auto-configures some default exception handling. It is important to have a consistent exception message which is obtained for all the services inside our enterprise.

If we have a big organization and each of the services returns the exception messages in a different way, so it is not good. It would be good if we define a standard exception structure which is followed by across all the RESTful Services.

Implementing Generic Exception Handling for all Resources

As we discussed earlier that we should define a standard exception structure which is followed across all the RESTful services. In this section, we will discuss the implementation **of generic exception handling for all resources**.

Let's see how to customize the exception message.

**Step 1:** Create a new package with name **com.javatpoint.server.main.exception**.

**Step 2:** Create a class with the name **ExceptionResponse** in the above package.

25.4M

526

HTML Tutorial

**Step 3:** At the basic level, there are three crucial things for the exception structure: **timestamp, message,** and **detail**. Define these three fields.

**Step 4:** Generate Constructors using Fields.

**Step 5:** Generate Getters, Setters are not needed.

Once the structure is defined, we can define Java implementation for it. Whenever an exception occurs, we would return a response in the specific format. The structure is the most important part and must have to be language independent.

One of the important class in Spring Framework is **ResponseEntityExceptionHandler** class. It is an abstract and base class for the exceptions that provide centralized exception handling across all the different exception handler methods. We will extend this class to handle and provide customized exception handling functionality. This exception handling functionality is applied to all controller as HellowWorldController, UserResource (Controller).

**Step 6:** Create a new class with name **CustomizedResponseEntityExceptionHandler** in the package **com.javatpoint.server.main.exception** and extends **ResponseEntityExceptionHandler** class.

**Step 7:** Add @ControllerAdvice and @RestController annotations.

**Step 8:** Expand Maven Dependencies in the package explorer -> Expand spring-webmvc-5.1.9.RELEASE.jar -> Expand org.springframework.web.servlet.mvc.method.annotation -> Open **ResponseEntityExceptionHandler.class**.

Implementing Generic Exception Handling for all Resources

**Step 9:** Copy the **ResponseEntityMethod<Object>** method from ResponseEntityExceptionHandler.class and paste in **CustomizedResponseEntityExceptionHandler.java** file.

**Step 10:** Override the **ResponseEntityMethod** method. Rename the method name as handleAllExceptions().

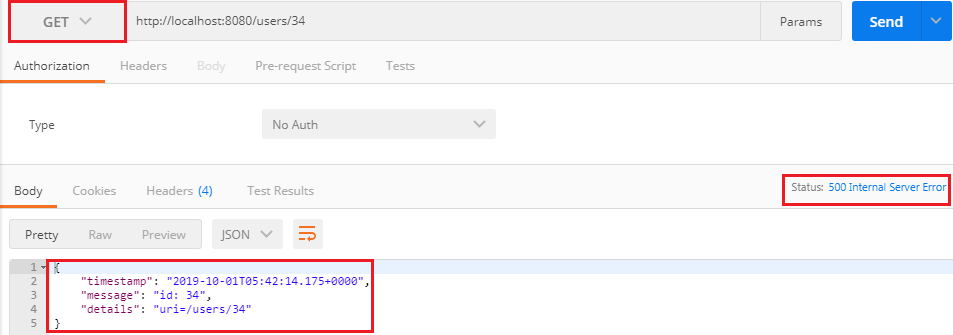
**Step 11:** Create the exception response structure.

**Step 12:** Create a **ResponseEntity** object and pass the exception response and Http status as arguments.

**CustomizedResponseEntityExceptionHandler.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.Date;
3. **import** org.springframework.http.HttpStatus;
4. **import** org.springframework.http.ResponseEntity;
5. **import** org.springframework.web.bind.annotation.ControllerAdvice;
6. **import** org.springframework.web.bind.annotation.ExceptionHandler;
7. **import** org.springframework.web.bind.annotation.RestController;
8. **import** org.springframework.web.context.request.WebRequest;
9. **import** org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
10. **import** com.javatpoint.server.main.exception.ExceptionResponse;
11. //defining exception handling for all the exceptions
12. @ControllerAdvice
13. @RestController
14. **public** **class** CustomizedResponseEntityExceptionHandler **extends** ResponseEntityExceptionHandler
15. {
16. @ExceptionHandler(Exception.**class**)
17. //override method of ResponseEntityExceptionHandler class
18. **public** **final** ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request)
19. {
20. //creating exception response structure
21. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));
22. //returning exception structure and specific status
23. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);
24. }
25. }

**Step 13:** Open rest client **Postman** and send a **Get** request. We get the **Status: 500 Internal Server Error** and exception structure which we have defined.

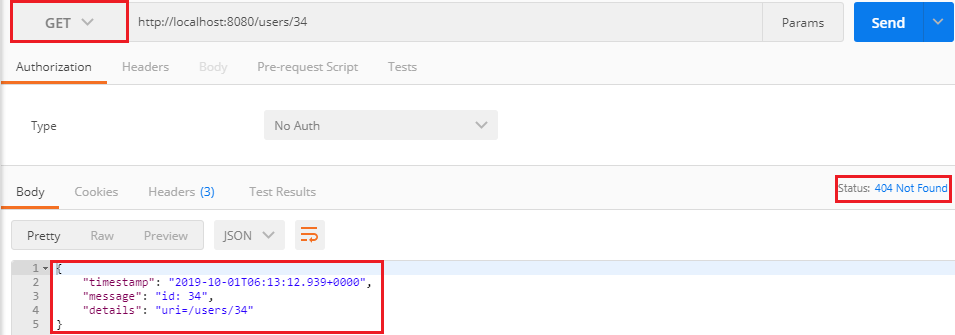


If we want to customize the status **Internal Server Error to Not Found**, we are required to change a few things in **CustomizedResponseEntityExceptionHandler.java** file.

**CustomizedResponseEntityExceptionHandler.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.Date;
3. **import** org.springframework.http.HttpStatus;
4. **import** org.springframework.http.ResponseEntity;
5. **import** org.springframework.web.bind.annotation.ControllerAdvice;
6. **import** org.springframework.web.bind.annotation.ExceptionHandler;
7. **import** org.springframework.web.bind.annotation.RestController;
8. **import** org.springframework.web.context.request.WebRequest;
9. **import** org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
10. **import** com.javatpoint.server.main.exception.ExceptionResponse;
11. **import** com.javatpoint.server.main.user.UserNotFoundException;
12. //defining exception handling for all the exceptions
13. @ControllerAdvice
14. @RestController
15. **public** **class** CustomizedResponseEntityExceptionHandler **extends** ResponseEntityExceptionHandler
16. {
17. @ExceptionHandler(Exception.**class**)
18. //override method of ResponseEntityExceptionHandler class
19. **public** **final** ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request)
20. {
21. //creating exception response structure
22. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));
23. //returning exception structure and Internal Server status
24. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);
25. }
26. @ExceptionHandler(UserNotFoundException.**class**)
27. //override method of ResponseEntityExceptionHandler class
28. **public** **final** ResponseEntity<Object> handleUserNotFoundExceptions(UserNotFoundException ex, WebRequest request)
29. {
30. //creating exception response structure
31. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));
32. //returning exception structure and Not Found status
33. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.NOT\_FOUND);
34. }
35. }

Again move to **Postman** and send a **Get** request. We get the **Status: 404 Not Found** with the defined exception structure.



Implementing DELETE Method to Delete a User Resource

In this section, we will implement a delete method to delete a user resource.

**Step 1:** Open the **UserDaoService.java** file.

**Step 2:** Create a method to delete a user resource.

UserDaoService.java

Difference between JDK, JRE, and JVM

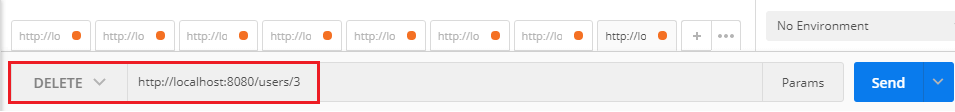
1. **package** com.javatpoint.server.main.user;
2. **import** java.util.ArrayList;
3. **import** java.util.Date;
4. **import** java.util.Iterator;
5. **import** java.util.List;
6. **import** org.springframework.stereotype.Component;
7. @Component
8. **public** **class** UserDaoService
9. {
10. **public** **static** **int** usersCount=5;
11. //creating an instance of ArrayList
12. **private** **static** List<User> users=**new** ArrayList<>();
13. //static block
14. **static**
15. {
16. //adding users to the list
17. users.add(**new** User(1, "John", **new** Date()));
18. users.add(**new** User(2, "Robert", **new** Date()));
19. users.add(**new** User(3, "Adam", **new** Date()));
20. users.add(**new** User(4, "Andrew", **new** Date()));
21. users.add(**new** User(5, "Jack", **new** Date()));
22. }
23. //method that retrieve all users from the list
24. **public** List<User> findAll()
25. {
26. **return** users;
27. }
28. //method that adds a user in the list
29. **public** User save(User user)
30. {
31. **if**(user.getId()==**null**)
32. {
33. user.setId(++usersCount);
34. }
35. users.add(user);
36. **return** user;
37. }
38. //method that find a particular user from the list
39. **public** User findOne(**int** id)
40. {
41. **for**(User user:users)
42. {
43. **if**(user.getId()==id)
44. **return** user;
45. }
46. **return** **null**;
47. }
48. //method that delete a user resource
49. **public** User deleteById(**int** id)
50. {
51. Iterator<User> iterator = users.iterator();
52. **while**(iterator.hasNext())
53. {
54. User user=iterator.next();
55. **if**(user.getId()==id)
56. {
57. iterator.remove();
58. **return** user; //returns the deleted resource back
59. }
60. }
61. **return** **null**;
62. }
63. }

**Step 3:** Open the UserResource.java file and create a delete mapping to delete a user resource.

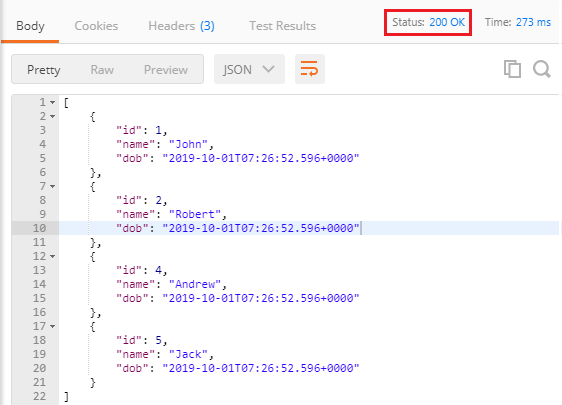
**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.net.URI;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.http.ResponseEntity;
6. **import** org.springframework.web.bind.annotation.DeleteMapping;
7. **import** org.springframework.web.bind.annotation.GetMapping;
8. **import** org.springframework.web.bind.annotation.PathVariable;
9. **import** org.springframework.web.bind.annotation.PostMapping;
10. **import** org.springframework.web.bind.annotation.RequestBody;
11. **import** org.springframework.web.bind.annotation.RestController;
12. **import** org.springframework.web.servlet.support.ServletUriComponentsBuilder;
13. @RestController
14. **public** **class** UserResource
15. {
16. @Autowired
17. **private** UserDaoService service;
18. @GetMapping("/users")
19. **public** List<User> retriveAllUsers()
20. {
21. **return** service.findAll();
22. }
23. //retrieves a specific user detail
24. @GetMapping("/users/{id}")
25. **public** User retriveUser(@PathVariable **int** id)
26. {
27. User user= service.findOne(id);
28. **if**(user==**null**)
29. //runtime exception
30. **throw** **new** UserNotFoundException("id: "+ id);
31. **return** user;
32. }
33. //method that delete a user resource
34. //if the user deleted successfully it returns status 200 OK otherwise 404 Not Found
35. @DeleteMapping("/users/{id}")
36. **public** **void** deleteUser(@PathVariable **int** id)
37. {
38. User user= service.deleteById(id);
39. **if**(user==**null**)
40. //runtime exception
41. **throw** **new** UserNotFoundException("id: "+ id);
42. }
43. //method that posts a new user detail and returns the status of the user resource
44. @PostMapping("/users")
45. **public** ResponseEntity<Object> createUser(@RequestBody User user)
46. {
47. User sevedUser=service.save(user);
48. URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(sevedUser.getId()).toUri();
49. **return** ResponseEntity.created(location).build();
50. }
51. }

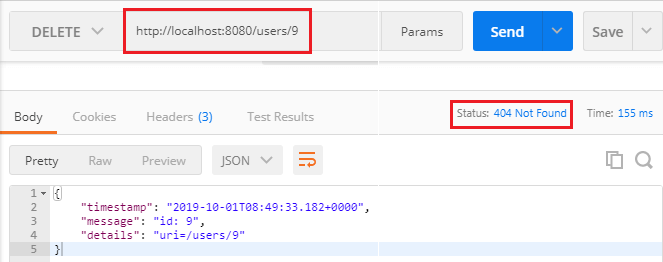
**Step 4:** Open **Postman**, select **DELETE** request, and specify the user **id** which you want to delete. Now click on the **Send** button.



It deletes user id: 3 and returns the **Status: 200 Ok**. Again send the **Get** request. It shows all users except user 3.



In the following image, we are trying to delete user id: 9, which does not exist. Hence it returns the **Status: 404 Not Found**.



Implementing Validations for RESTful Services

The validation is a common requirement in all the services. We will discuss Java Validation API to add validation in our beans file. When we get a request to create a user, we should validate its content. If it is invalid, we should return a proper response.

Let's see how to validate a request.

**Step 1:** Open the **UserResource.java** file.

**Step 2:** Add **@Valid** annotation. It is a Javax validation API. Its default classpath is spring-boot-starter-web.

30M

694

OOPs Concepts in Java

**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.net.URI;
3. **import** java.util.List;
4. **import** javax.validation.Valid;
5. **import** org.springframework.beans.factory.annotation.Autowired;
6. **import** org.springframework.http.ResponseEntity;
7. **import** org.springframework.web.bind.annotation.DeleteMapping;
8. **import** org.springframework.web.bind.annotation.GetMapping;
9. **import** org.springframework.web.bind.annotation.PathVariable;
10. **import** org.springframework.web.bind.annotation.PostMapping;
11. **import** org.springframework.web.bind.annotation.RequestBody;
12. **import** org.springframework.web.bind.annotation.RestController;
13. **import** org.springframework.web.servlet.support.ServletUriComponentsBuilder;
14. @RestController
15. **public** **class** UserResource
16. {
17. @Autowired
18. **private** UserDaoService service;
19. @GetMapping("/users")
20. **public** List<User> retriveAllUsers()
21. {
22. **return** service.findAll();
23. }
24. //retrieves a specific user detail
25. @GetMapping("/users/{id}")
26. **public** User retriveUser(@PathVariable **int** id)
27. {
28. User user= service.findOne(id);
29. **if**(user==**null**)
30. //runtime exception
31. **throw** **new** UserNotFoundException("id: "+ id);
32. **return** user;
33. }
34. //method that delete a user resource
35. @DeleteMapping("/users/{id}")
36. **public** **void** deleteUser(@PathVariable **int** id)
37. {
38. User user= service.deleteById(id);
39. **if**(user==**null**)
40. //runtime exception
41. **throw** **new** UserNotFoundException("id: "+ id);
42. }
43. //method that posts a new user detail and returns the status of the user resource
44. @PostMapping("/users")
45. **public** ResponseEntity<Object> createUser(@Valid @RequestBody User user)
46. {
47. User sevedUser=service.save(user);
48. URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(sevedUser.getId()).toUri();
49. **return** ResponseEntity.created(location).build();
50. }
51. }

Now we will add validations in **User** class on **name** and **date of birth**. Let suppose that name should have at least five characters and date of birth should be in past not in present.

**Step 3:** Open the **User.java** file.

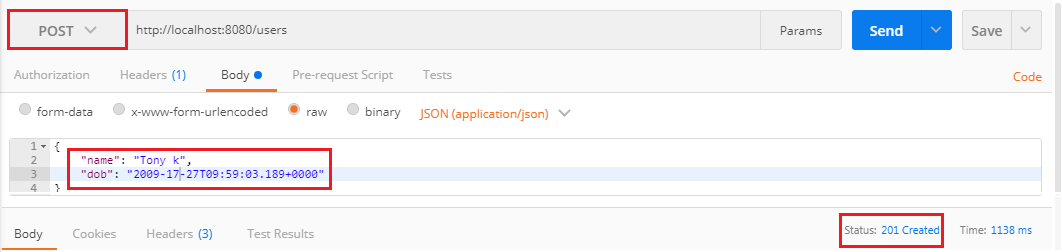
**Step 4:** Add **@Size(min=5)** annotation just above the **name** variable.

**Step 5:** Add **@Past** annotation just above the **dob** variable.

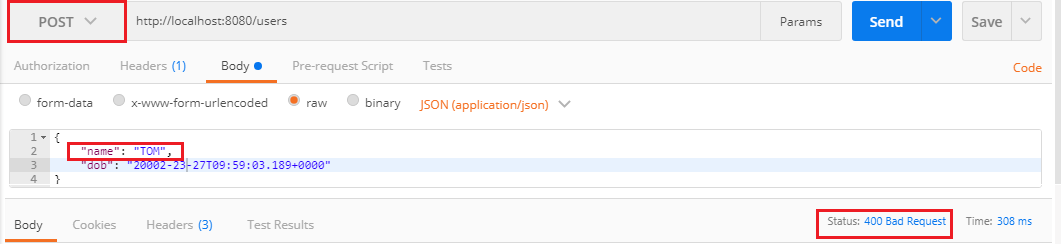
**User.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.Date;
3. **import** javax.validation.constraints.Past;
4. **import** javax.validation.constraints.Size;
5. **public** **class** User
6. {
7. **private** Integer id;
8. @Size(min=5)
9. **private** String name;
10. @Past
11. **private** Date dob;
12. //default constructor
13. **protected** User()
14. {
16. }
17. **public** User(Integer id, String name, Date dob)
18. {
19. **super**();
20. **this**.id = id;
21. **this**.name = name;
22. **this**.dob = dob;
23. }
24. **public** Integer getId()
25. {
26. **return** id;
27. }
28. **public** **void** setId(Integer id)
29. {
30. **this**.id = id;
31. }
32. **public** String getName()
33. {
34. **return** name;
35. }
36. **public** **void** setName(String name)
37. {
38. **this**.name = name;
39. }
40. **public** Date getDob()
41. {
42. **return** dob;
43. }
44. **public** **void** setDob(Date dob)
45. {
46. **this**.dob = dob;
47. }
48. @Override
49. **public** String toString()
50. {
51. //return "User [id=" + id + ", name=" + name + ", dob=" + dob + "]";
52. **return** String.format("User [id=%s, name=%s, dob=%s]", id, name, dob);
53. }
54. }

**Step 5:** Open the Rest client **Postman** and send a **POST** request with new user name **Tony k**. It returns **Status: 201 Created**.



Now we send another **POST** request. But the name should have less than five characters. It returns the **Status: 400 Bad Request**.



When we create a RESTful services we need to think about consumer that how does the consumer know what is wrong. To resolve this problem we will add a method handleMethodArgumentNotValid() which is defined in the **ResponseEntityExceptionHandler** class. This is the method which is fired when a bad request occurs.

1. **protected** ResponseEntity<Object> handleMethodArgumentNotValid(  MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request)
2. {
3. **return** handleExceptionInternal(ex, **null**, headers, status, request);
4. }

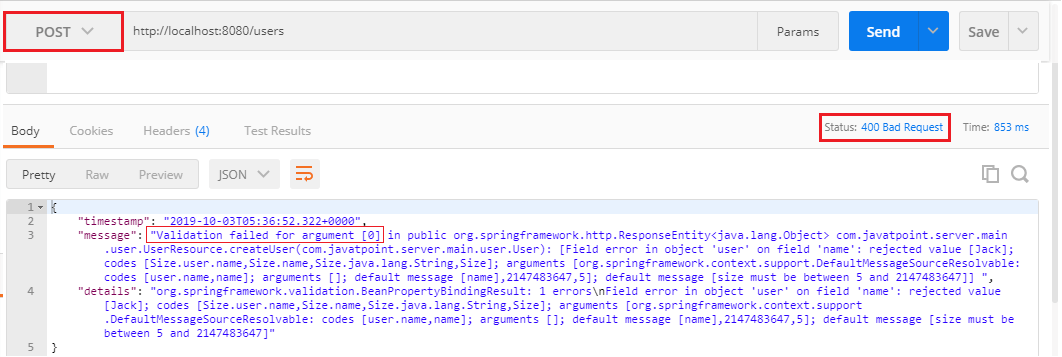
**Step 6:** Copy and paste the above method in **CustomizedResponseEntityExceptionHandler.java** file.

**Step 7:** Override the method by adding the annotation **@Override**.

**CustomizedResponseEntityExceptionHandler.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.Date;
3. **import** org.springframework.http.HttpHeaders;
4. **import** org.springframework.http.HttpStatus;
5. **import** org.springframework.http.ResponseEntity;
6. **import** org.springframework.web.bind.MethodArgumentNotValidException;
7. **import** org.springframework.web.bind.annotation.ControllerAdvice;
8. **import** org.springframework.web.bind.annotation.ExceptionHandler;
9. **import** org.springframework.web.bind.annotation.RestController;
10. **import** org.springframework.web.context.request.WebRequest;
11. **import** org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
12. **import** com.javatpoint.server.main.exception.ExceptionResponse;
13. **import** com.javatpoint.server.main.user.UserNotFoundException;
14. //defining exception handling for all the exceptions
15. @ControllerAdvice
16. @RestController
17. **public** **class** CustomizedResponseEntityExceptionHandler **extends** ResponseEntityExceptionHandler
18. {
19. @ExceptionHandler(Exception.**class**)
20. //override method of ResponseEntityExceptionHandler class
21. **public** **final** ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request)
22. {
23. //creating exception response structure
24. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));
25. //returning exception structure and specific status
26. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);
27. }
28. @ExceptionHandler(UserNotFoundException.**class**)
29. //override method of ResponseEntityExceptionHandler class
30. **public** **final** ResponseEntity<Object> handleUserNotFoundExceptions(UserNotFoundException ex, WebRequest request)
31. {
32. //creating exception response structure
33. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));
34. //returning exception structure and specific status
35. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.NOT\_FOUND);
36. }
37. @Override
38. **protected** ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request)
39. {
40. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), ex.getBindingResult().toString());
41. //returning exception structure and specific status
42. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.BAD\_REQUEST);
43. }
44. }

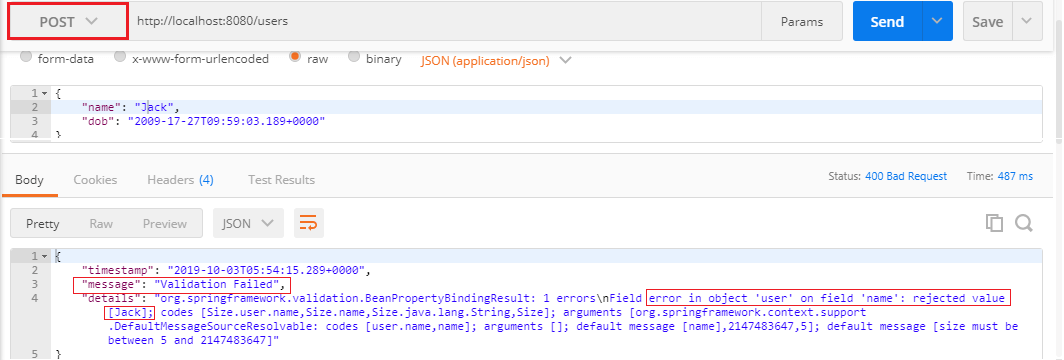
**Step 8:** Now, we send a **POST** request through **Postman**. It returns the exception structure with message **Validation failed for argument** and other details.



It is difficult to understand the message for the user. So we will now customize the message with the string **Validation Failed** instead of getting message.

1. @Override
2. **protected** ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request)
3. {
4. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), "Validation Failed", ex.getBindingResult().toString());
5. //returning exception structure and specific status
6. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.BAD\_REQUEST);
7. }

**Step 9:** Again send a **POST** request. It returns the message which we have customized.

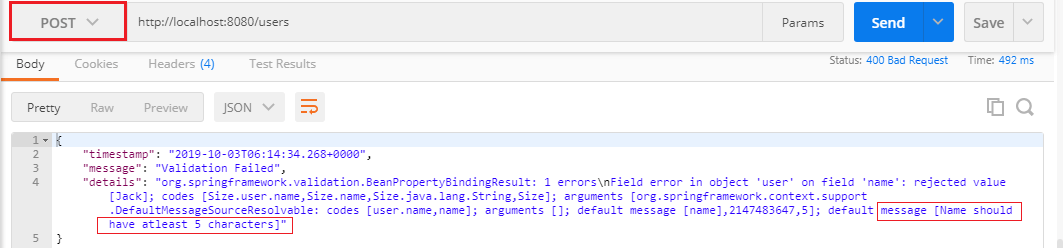


It might be useful for the consumer. Now we customize the message again and make it more error specific.

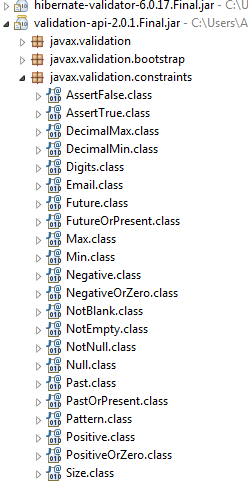
**Step 10:** Open **User.java** file and add an attribute **message="Name should have at least 5 characters"** in **@Size** annotation.

1. @Size(min=5, message="Name should have at least 5 characters")

**Step 11:** Again, send a **POST** request. It returns the more specific exception which we have specified.



We can further customize the exception by following the **BindingResult** interface. There is a wide variety of exception messages. There are following validation classes defined in validation-api-2.0.1.Final.jar.



# Implementing HATEOAS for RESTful Services

## HATEOAS

HATEOAS acronyms for **Hypermedia as the Engine of Application State**. The term hypermedia refers to content that contains a link to other forms of media like images, movies, and text. It is a component of the REST application that distinguishes it from other network architecture. Using HATEOAS, a client interacts with a network application, whose application server provides information dynamically through Hypermedia.

### **Spring-HATEOAS**

Spring-HATEOAS is the library of APIs. We can use these APIs for creating REST representations that follow the HATEOAS principle while working with Spring MVC.

In the Spring HATEOAS project, we do not require Servlet Context and concatenate the path variable to the base URI. Instead of this, Spring HATEOAS offers three abstractions for creating the URI: **ContrrollerLinkBuilder, Link,** and **Resource Support**. We can use these abstractions to create metadata, which associates with the resource representation.

**Features**

30M

694

OOPs Concepts in Java

* It supports **hypermedia** formats like HAL.
* It provides **Link builder API** to create links pointing to MVC controller methods.
* **Model classes** for the link, resource representation models.

Spring Boot does the following tasks:

* Configure HAL support
* Register support for entity links
* Wire up message converter support

Suppose, we have requested a GET request for **localhost:8080/users/1**, it returns the details of user id 1. Along with this, it also returns a field called **link** that contains a link (**localhost:8080/users**) of all users so that consumers can retrieve all the users. This concept is called **HATEOAS**.

Let's implement the HATEOAS in the project.

**Step1:** Open the **pom.xml** and add the **spring-boot-starter-hateoas** dependency.

1. <dependency>d
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-hateoas</artifactId>
4. </dependency>

**Step 2:** Open **UserResource.java** and copy the retrieveUser() method.

**Step 3:** Paste the method and make the following changes:

* Create a constructor of Resource class.

1. Resource<User> resource = **new** Resource<User>(User)

Remember that import the **Resource** class of **org.springframework.hateoas** package.

* Add a link to retrieve all users by using the **ControllerLinkBuilder** class. It enables us to create a link from methods.
* Import the ControllerLinkBuilder.

1. **import** **static** org.springframework.hateoas.mvc.ControllerLinkBuilder.

* Use the method **linkTo()** of ControllerLinkBuilder class. It creates a new ControllerLinkBuilder with a base of mapping annotated to the given controller class.

1. ControllerLinkBuilder linkTo=linkTo(methodOn(**this**.getClass().retrieveAllUsers());

methodOn() is a wrapper for **DummyInvocationUtils.methodOn(class, Object)** to be available in case you work with the static import of ControllerLinkBuilder.

* Add this link to the resource with the name which we want to use inside the HATEOAS.

1. resource.add(linkTo.withRel("all-users"));

**withRel(String rel)** is the method that creates the link built by the current builder instance with the given rel. The parameter rel must not be null.

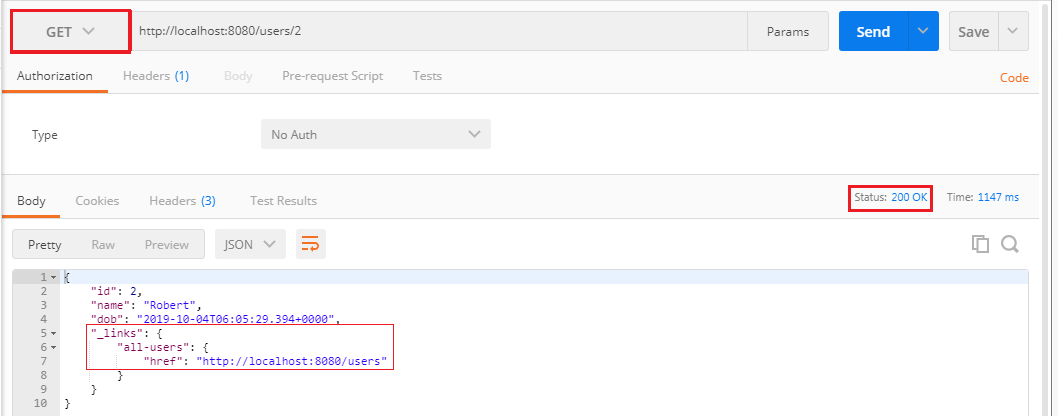
* Return the **resource** instead of the user.
* Change the return type of method to **Resource**.

After making the above changes the UserResource.java file look like the following:

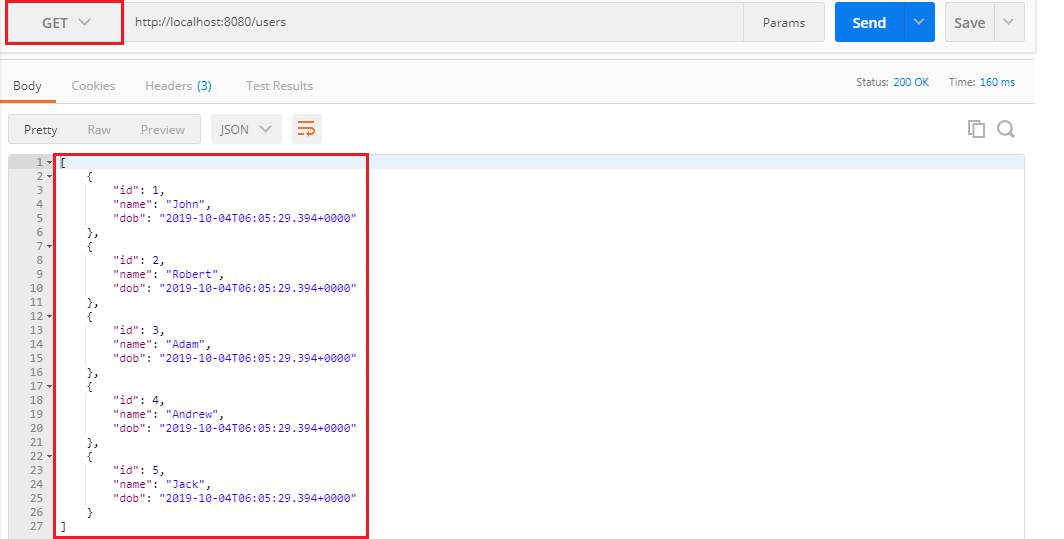
**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** **static** org.springframework.hateoas.mvc.ControllerLinkBuilder.\*;
3. **import** java.net.URI;
4. **import** java.util.List;
5. **import** javax.validation.Valid;
6. **import** org.springframework.beans.factory.annotation.Autowired;
7. **import** org.springframework.hateoas.Resource;
8. **import** org.springframework.hateoas.mvc.ControllerLinkBuilder;
9. **import** org.springframework.http.ResponseEntity;
10. **import** org.springframework.web.bind.annotation.DeleteMapping;
11. **import** org.springframework.web.bind.annotation.GetMapping;
12. **import** org.springframework.web.bind.annotation.PathVariable;
13. **import** org.springframework.web.bind.annotation.PostMapping;
14. **import** org.springframework.web.bind.annotation.RequestBody;
15. **import** org.springframework.web.bind.annotation.RestController;
16. **import** org.springframework.web.servlet.support.ServletUriComponentsBuilder;
17. @RestController
18. **public** **class** UserResource
19. {
20. @Autowired
21. **private** UserDaoService service;
22. @GetMapping("/users")
23. **public** List<User> retriveAllUsers()
24. {
25. **return** service.findAll();
26. }
27. @GetMapping("/users/{id}")
28. **public** Resource<User> retriveUser(@PathVariable **int** id)
29. {
30. User user= service.findOne(id);
31. **if**(user==**null**)
32. //runtime exception
33. **throw** **new** UserNotFoundException("id: "+ id);
34. //"all-users", SERVER\_PATH + "/users"
35. //retrieveAllUsers
36. Resource<User> resource=**new** Resource<User>(user);   //constructor of Resource class
37. //add link to retrieve all the users
38. ControllerLinkBuilder linkTo=linkTo(methodOn(**this**.getClass()).retriveAllUsers());
39. resource.add(linkTo.withRel("all-users"));
40. **return** resource;
41. }
42. //method that delete a user resource
43. @DeleteMapping("/users/{id}")
44. **public** **void** deleteUser(@PathVariable **int** id)
45. {
46. User user= service.deleteById(id);
47. **if**(user==**null**)
48. //runtime exception
49. **throw** **new** UserNotFoundException("id: "+ id);
50. }
51. //method that posts a new user detail and returns the status of the user resource
52. @PostMapping("/users")
53. **public** ResponseEntity<Object> createUser(@Valid @RequestBody User user)
54. {
55. User sevedUser=service.save(user);
56. URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(sevedUser.getId()).toUri();
57. **return** ResponseEntity.created(location).build();
58. }
59. }

**Step 4:** Open the REST client **Postman** and send a **GET** request.



Here we can see that it returns the user along with the link to access all-users. Now click on the link and send a GET request again. It returns the list of all users, as shown in the following image.



# Internationalization of RESTful Services

In this section, we will discuss the Internationalization of the RESTful Web Services.

## Internationalization

Internationalization is the process of designing web applications or services in such a way that it can provide support for various countries, various languages automatically without making the changes in the application. It is also known as **I18N** because the word internationalization has total 18 characters starting from **I** to **N**.

Localization is performed by adding locale-specific components such as translated text, data describing locale-specific behavior, etc. It supports full integration into the classes and packages that provide language-or-culture-dependent functionality.

Java provides the foundation for internationalization for desktop and server applications. There are following important internationalized areas of functionality.

33M

569

Hello Java Program for Beginners

* **Text Representation:** Java is based on the Unicode character set, and several libraries implement the Unicode standard.
* **Locale identification and localization:** Locale in Java are identifiers that can be used to request locale-specific behavior in different areas of functionality. Localization is supported by ResourceBundle class. The class provides access to local specific objects, including strings.
* **Date and time handling:** Java provides various calendars. It supports conversion to and from calendar independent Date objects. Java supports all the time zones in the world.
* **Text processing:** It includes character analysis, case mapping, string comparison, breaking text into words, formatting numbers, dates, and time values into strings or parsing them back from strings. Most of these functions are locale-dependent.
* **Character encoding:** It supports converting text between Unicode and other character encodings when reading incoming text from the streams or writing outgoing text to the streams.

We need to configure two things to make the service internationalized.

* LocaleResolver
* ResourceBundleMessageSource

Default Locale is Locale.US. If somebody does not specify the location, it returns the default locale. We also need to customize the ResourceBundle. It has a list of properties that are to be internationalized. We will store the properties in ResourceBundle. **ResourceBundleMessageSource** is a Spring MVC concept for handling properties. After that, we will use MessageSource, and a header called Accept-Language.

Let's configure the internationalization.

**Step 1:** Open **RestfulWebServicesApplication.java** file.

**Step 2:** Configure a Bean for **default** locale.

1. @Bean
2. **public**  LocaleResolver localeResolver()
3. {
4. SessionLocaleResolver localeResolver = **new** SessionLocaleResolver();
5. localeResolver.setDefaultLocale(Locale.US);
6. **return** localeResolver;
7. }

#### **Note: Import import org.springframework.web.servlet.LocaleResolver package while importing LocaleResolver.**

**Step 3:** Now, we will store properties in a specific file called **messages.properties**.

Right-click on **src/main/resources** folder -> New -> File -> Provide the file name: **messages.properties**. It contains the default locale message.

**messages.properties**

1. good.morning.message=Good Morning

**Step 4:** Create another property file with the name **messages\_fr.properties** for French locale. It contains a message for the French locale.

**messages\_fr.properties**

1. good.morning.message=Bonjour

**Step 5:** Read properties and customize them based on the input accept header. Open the RestfulWebServicesApplication.java and configure another Bean for **ResourceBundle**.

1. //configuring ResourceBundle
2. @Bean
3. **public** ResourceBundleMessageSource bundleMessageSource()
4. {
5. ResourceBundleMessageSource messageSource = **new** ResourceBundleMessageSource();
6. messageSource.setBasename("messages");
7. **return** messageSource;
8. }

**RestfulWebServicesApplication.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.Locale;
3. **import** org.springframework.boot.SpringApplication;
4. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
5. **import** org.springframework.context.annotation.Bean;
6. **import** org.springframework.context.support.ResourceBundleMessageSource;
7. **import** org.springframework.web.servlet.LocaleResolver;
8. **import** org.springframework.web.servlet.i18n.SessionLocaleResolver;
9. @SpringBootApplication
10. **public** **class** RestfulWebServicesApplication
11. {
12. **public** **static** **void** main(String[] args)
13. {
14. SpringApplication.run(RestfulWebServicesApplication.**class**, args);
15. }
16. //configuring default locale
17. @Bean
18. **public**  LocaleResolver localeResolver()
19. {
20. SessionLocaleResolver localeResolver = **new** SessionLocaleResolver();
21. localeResolver.setDefaultLocale(Locale.US);
22. **return** localeResolver;
23. }
24. //configuring ResourceBundle
25. @Bean
26. **public** ResourceBundleMessageSource messageSource()
27. {
28. ResourceBundleMessageSource messageSource = **new** ResourceBundleMessageSource();
29. messageSource.setBasename("messages");
30. **return** messageSource;
31. }
32. }

**Step 6:** Update the service to use these sources. Open the **HelloWorldController.java** and autowired the MessageSource.

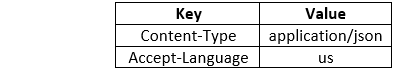
1. @Autowired
2. **private** MessageSource messageSource;

**HelloWorldController.java**

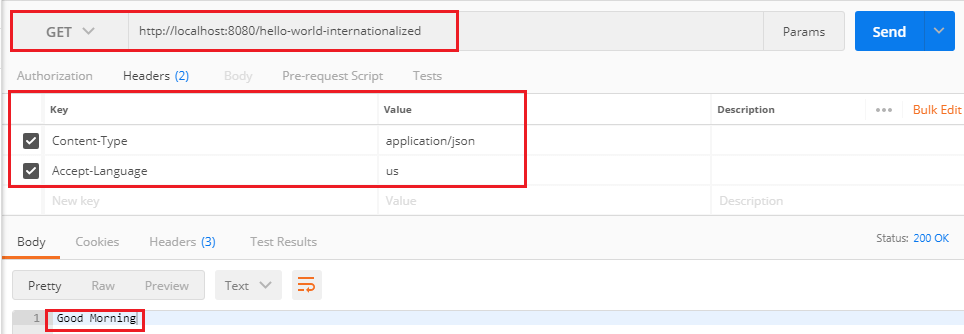
1. **package** com.javatpoint.server.main.helloworld;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.PathVariable;
4. **import** org.springframework.web.bind.annotation.RequestHeader;
5. **import** org.springframework.web.bind.annotation.RestController;
6. **import** java.util.Locale;
7. **import** org.springframework.beans.factory.annotation.Autowired;
8. **import** org.springframework.context.MessageSource;
9. **import** org.springframework.context.annotation.Configuration;
10. @Configuration
11. //Controller
12. @RestController
13. **public** **class** HelloWorldController
14. {
15. @Autowired
16. **private** MessageSource messageSource;
17. //using get method and hello-world URI
18. @GetMapping(path="/hello-world")
19. **public** String helloWorld()
20. {
21. **return** "Hello World";
22. }
23. @GetMapping(path="/hello-world-bean")
24. //method- which returns "Hello World"
25. **public** HelloWorldBean helloWorldBean()
26. {
27. **return** **new** HelloWorldBean("Hello World");//constructor of HelloWorldBean
28. }
29. //passing a path variable
30. //hello-world/path-variable/javatpoint
31. @GetMapping(path="/hello-world/path-variable/{name}")
32. **public** HelloWorldBean helloWorldPathVariable(@PathVariable String name)
33. {
34. **return** **new** HelloWorldBean(String.format("Hello World, %s",name));   //%s replace the name
35. }
36. //internationalization
37. @GetMapping(path="/hello-world-internationalized")
38. **public** String helloWorldInternationalized(@RequestHeader(name="Accept-Language", required=**false**) Locale locale)
39. {
40. **return** messageSource.getMessage("good.morning.message", **null**, locale);
41. }
42. }

**Step 7:** Open the REST client **Postman** and perform the following changes:

* Select the **GET** request.
* Type the URI http://localhost:8080/hello-world-internationalized
* Click on **Headers** tab and type:

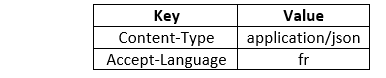
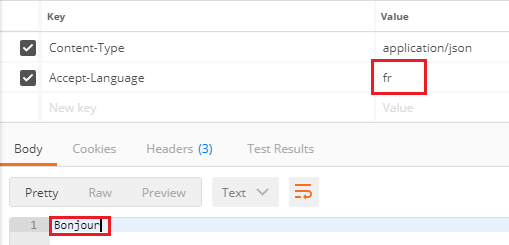


* Click on the Send button.



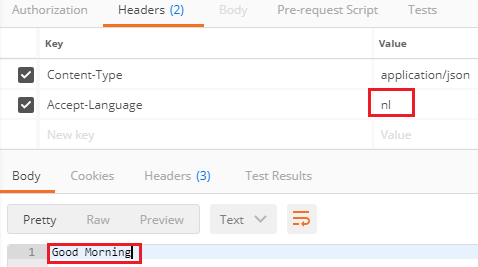
It returns the US locale message **Good Morning**.

Now we change the RequestHeader us to fr and send a GET request again.

It returns the French locale message **Bonjour**.

Again, change the RequestHeader fr to other RequestHeader, say **nl**. It returns the default locale (US) message **Good Morning**.

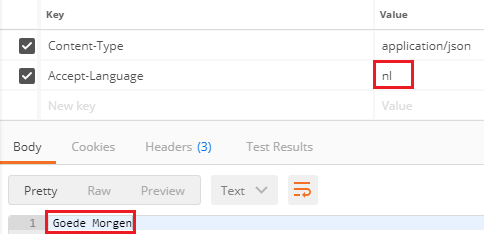


Let's create a property file **message\_nl.properties** for RequestHeader **nl**. It contains a message Goede Morgen in the **Dutch** language.

**messages\_nl.properties**

1. good.morning.message=Goede Morgen

Again send a **GET** request that returns the message, **Goede Morgen**.



## Simplify the Internationalization

Now we will simplify the implementation of internationalization, which we have done above. In the previous implementation, we have accepted locale (a RequestHeader) as a parameter to the REST controller method. If we add this to every method that has to be internationalized, it will increase the cost. Spring provides an alternative way to get it from the **LocaleContextHolder**.

Let's implement the LocaleContextHolder instead of RequestHeader.

**Step 1:** Open the **HelloWorldController.java** and change the return type of the helloWorldInternationalized() method.

1. **return** messageSource.getMessage("good.morning.message", **null**, LocaleContextHolder.getLocale());

**HelloWorldController.java**

1. **package** com.javatpoint.server.main.helloworld;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.PathVariable;
4. **import** org.springframework.web.bind.annotation.RequestHeader;
5. **import** org.springframework.web.bind.annotation.RestController;
6. **import** java.util.Locale;
7. **import** org.springframework.beans.factory.annotation.Autowired;
8. **import** org.springframework.context.MessageSource;
9. **import** org.springframework.context.annotation.Configuration;
10. **import** org.springframework.context.i18n.LocaleContextHolder;
11. @Configuration
12. //Controller
13. @RestController
14. **public** **class** HelloWorldController
15. {
16. @Autowired
17. **private** MessageSource messageSource;
18. //using get method and hello-world URI
19. @GetMapping(path="/hello-world")
20. **public** String helloWorld()
21. {
22. **return** "Hello World";
23. }
24. @GetMapping(path="/hello-world-bean")
25. //method- which returns "Hello World"
26. **public** HelloWorldBean helloWorldBean()
27. {
28. **return** **new** HelloWorldBean("Hello World");//constructor of HelloWorldBean
29. }
30. //passing a path variable
31. //hello-world/path-variable/javatpoint
32. @GetMapping(path="/hello-world/path-variable/{name}")
33. **public** HelloWorldBean helloWorldPathVariable(@PathVariable String name)
34. {
35. **return** **new** HelloWorldBean(String.format("Hello World, %s",name));   //%s replace the name
36. }
37. //internationalization
38. @GetMapping(path="/hello-world-internationalized")
39. **public** String helloWorldInternationalized(@RequestHeader(name="Accept-Language", required=**false**) Locale locale)
40. {
41. **return** messageSource.getMessage("good.morning.message", **null**, LocaleContextHolder.getLocale());
42. }
43. }

**Step 2:** Open **RestfulWebServicesApplication.java** and change SessionLocaleResolver to AcceptHeaderLocaleResolver. LocaleResolver implementation uses the primary locale specified in the "accept-language" header of the HTTP request (locale send by the client browser).

**RestfulWebServicesApplication.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.Locale;
3. **import** org.springframework.boot.SpringApplication;
4. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
5. **import** org.springframework.context.annotation.Bean;
6. **import** org.springframework.context.support.ResourceBundleMessageSource;
7. **import** org.springframework.web.servlet.LocaleResolver;
8. **import** org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver;
9. **import** org.springframework.web.servlet.i18n.SessionLocaleResolver;
10. @SpringBootApplication
11. **public** **class** RestfulWebServicesApplication
12. {
13. **public** **static** **void** main(String[] args)
14. {
15. SpringApplication.run(RestfulWebServicesApplication.**class**, args);
16. }
17. //configuring default locale
18. @Bean
19. **public**  LocaleResolver localeResolver()
20. {
21. AcceptHeaderLocaleResolver localeResolver = **new** AcceptHeaderLocaleResolver();
22. localeResolver.setDefaultLocale(Locale.US);
23. **return** localeResolver;
24. }
25. //configuring ResourceBundle
26. @Bean
27. **public** ResourceBundleMessageSource messageSource()
28. {
29. ResourceBundleMessageSource messageSource = **new** ResourceBundleMessageSource();
30. messageSource.setBasename("messages");
31. **return** messageSource;
32. }
33. }

The advantage of AcceptHeaderLocaleResolver is that we do not require to configure request header as a parameter in every controller method.

**Step 3:** Open the REST client Postman and send a GET request by setting the key as Accept-Language and value as fr. It returns the message Bonjour.

Now uncheck the RequestHeader, change the value fr to en. It returns the default locale (US) message Good Morning.

**Step 4:** Move to RestfulWebServicesApplication.java. Remove the ResourceBundleMessageSource() method and configure it in the **application.properties** file.

**Step 5:** Open application.properties file and configure the message basename instead of creating a separate bean in RestfulWebServicesApplication.java.

**application.properties**

1. logging.level.org.springframework=info
2. spring.messages.basename=messages

**Step 6:** Repeat step 3.

Content Negotiation Implementing Support for XML

In this section, we will discuss another concept of RESTful Web Services that is content negotiation.

Content Negotiation

A resource can have many representations, mostly because there may be multiple clients expecting different representations.

Content negotiation is the process of selecting the best representation for a given response when there are multiple representations available. It is a part of HTTP that makes it possible to serve different versions of a document at the same URI.

In web API, content negotiation is performed at the server side to determine the media type formatted to be used based on return the response for an incoming request from the client-side. Content negotiation is centered on the media type and media type formatter.

32.2M

739

Features of Java - Javatpoint

Server-driven vs. Agent-driven Content Negotiation

An algorithm located at the server makes the selection of representation for a response, which is called **server-driven negotiation**.

An agent makes the selection of representation for a response, is called **agent-driven content negotiation**.

So, most of RSET API implementations rely on agent-driven content negotiations. Agent driven content negotiation rely on the usage of **HTTP requests** or **resource URI patterns**.

Content negotiation using HTTP headers

An incoming request may have an entity attached to it. To determine the type of entity server uses the HTTP request header **Content-Type**. There are some common content types are: application/json, application/xml, text/html, images/jpg, etc.

1. Content-Type: application/xml

HTTP header **ACCEPT** is used to determine what type of representation is desired at the client-side. It contains a value as mentioned for Content-Type.

1. Accept: application/json

If there is no header is present in the request, the server can send preconfigured default representation type.

Content negotiation using URL patterns

There is another way to pass content type information to the server. The client can use a specific extension in resource URIs. For example, a client can request for the following:

1. http://demo.com/product/mobile/samsung/galaxy-s8/functions.xml
2. http://demo.com/product/mobile/samsung/ galaxy-s8/functions.json

The first request URI returns the XML response, and the second URI returns the JSON response.

Defining Preferences

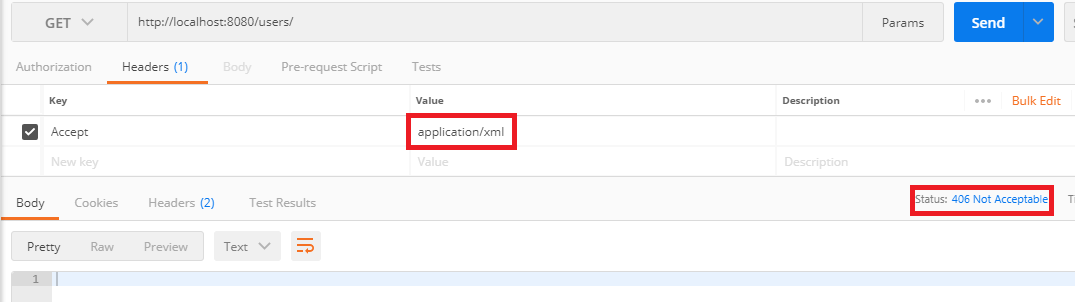
The preference is defined through the q parameter that has values between 0 and 1. If the client does not specify the request header, it takes the implicit value, i.e., 1.

If the client is not sure about its desired representation and wants to give multiple values in the accept header. For example:

1. Accept: application/json, application/xml;q=0.7,\*/\*;q=0.5

The above accept header allows you to ask the server for a JSON format. If the JSON format is not present, it will look for XML format. If the XML format is not possible, let it return what it can.

All services we are created until now only work with JSON input and JSON output. If we want to send a GET request by using HTTP header **application/xml**, it will return the **status: 406 Not Acceptable**.

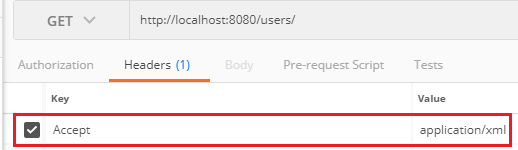


The above image shows that XML is not a valid accept header. Let's see how to implement XML format to one of the representations that are supported.

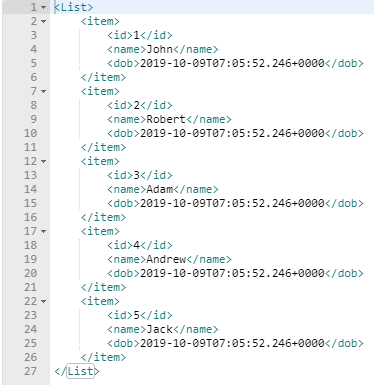
**Step 1:** Open **pom.xml** and add jackson-dataformat-xml dependency.

**Step 2:** Re-launch the application.

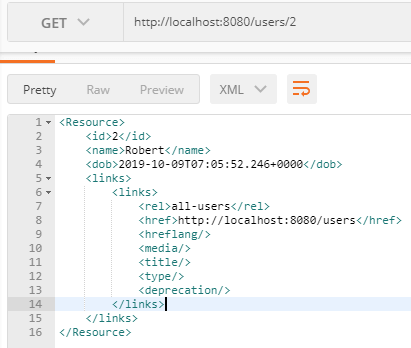
**Step 3:** Open the REST client **Postman** and send a **GET** request by specifying the HTTP header **Accept: application/xml**.



The following image shows the response in XML format.

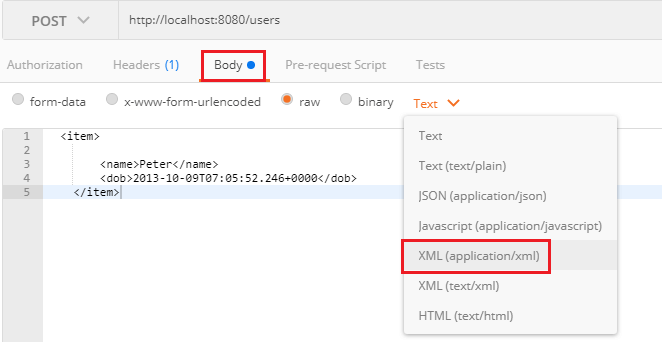


Similarly, we can send a GET request for a specific user.



Let's send a **POST** request using the same HTTP header.

* Under the Header tab add Content-Type: application/xml and Accept: application/xml.
* Select the **Body** tab.
* Enter the name and dob of the user in XML format.
* Select the XML format **XML (application/xml)**.
* Click on the **Send** button.



We get the **Status: 201 Created**. It means that user has been created successfully. Now it can support both the formats XML and JSON.

# Configuring Auto Generation of Swagger Documentation

## Swagger

A Swagger is an open-source tool. It builds around the OpenAPI Specification that helps developers to design, build, document, and consume RESTful APIs. It is the most popular API documentation format for RESTful Web Services. It provides both JSON and UI support. JSON can be used as a machine-readable format, and Swagger-UI is for visual display, which is easy for humans to understand by just browsing the API documentation. The main Swagger tools are:

* **Swagger UI:** It creates interactive API documentation.
* **Swagger Editor:** It is a browser-based editor where we can write OpenAPI specifications.
* **Swagger Codegen:** It generates server stubs (API implementation stub), and client libraries form an OpenAPI specification.

An **OpenAPI Specification** (formerly known as Swagger Specification) is an API documentation format for REST APIs. An Open API file allows us to describe our entire API, including:

* Available **endpoints (/users)** and **operations** on each endpoint (GET /users, POST /users).
* Operation parameters for each operation.
* Authentication methods.
* Contact information, license, the term of use and other information

**Let's generate Swagger documentation for our RESTful services.**

**Step 1:** Open **pom.xml** and add **springfox-swagger2** dependency.

33M

569

Hello Java Program for Beginners

1. <dependency>
2. <groupId>io.springfox</groupId>
3. <artifactId>springfox-swagger2</artifactId>
4. <version>2.9.2</version>
5. </dependency>

Add another dependency **springfox-swagger-ui**

1. <dependency>
2. <groupId>io.springfox</groupId>
3. <artifactId>springfox-swagger-ui</artifactId>
4. <version>2.9.2</version>
5. </dependency>

Now we need to configure Swagger.

**Step 2:** Create a class with the name **SwaggerConfig.java** and write the following code.

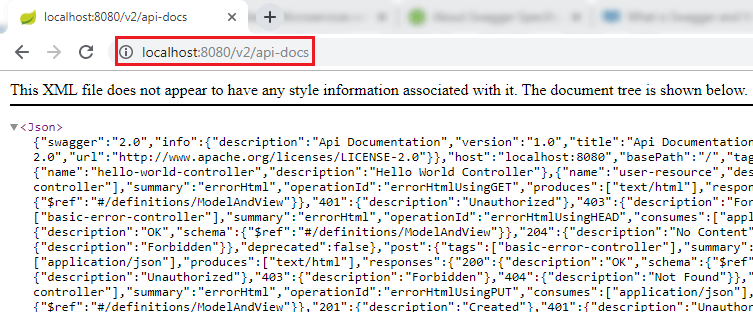
**Docket:** A builder that is intended to be the primary interface into the swagger-Spring MVC Framework. Docket provides sensible defaults and convenience methods for configuration.

1. **package** com.javatpoint.server.main;
2. **import** org.springframework.context.annotation.Bean;
3. **import** org.springframework.context.annotation.Configuration;
4. **import** springfox.documentation.spi.DocumentationType;
5. **import** springfox.documentation.spring.web.plugins.Docket;
6. **import** springfox.documentation.swagger2.annotations.EnableSwagger2;
7. @Configuration
8. //Enable Swagger
9. @EnableSwagger2
10. **public** **class** SwaggerConfig
11. {
12. //creating bean
13. @Bean
14. **public** Docket api()
15. {
16. //creating constructor of Docket class that accepts parameter DocumentationType
17. **return** **new** Docket(DocumentationType.SWAGGER\_2);
18. }
19. }

**Step 3:** Run the application.

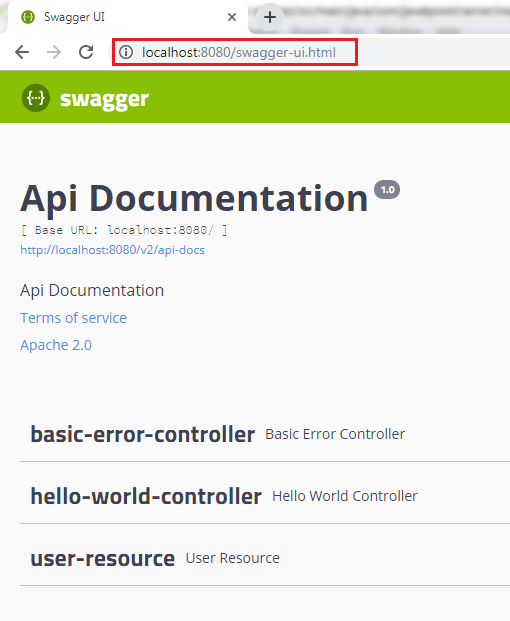
**Step 4:** Open the browser and type the URI http://localhost:8080/v2/api-docs

It shows the complete documentation in JSON format, as shown in the following image. It is not much easy to read and understand. Swagger has provided it to be used in other systems like API management tools that offer functionality like API gateways, API caching, API documentation, etc.

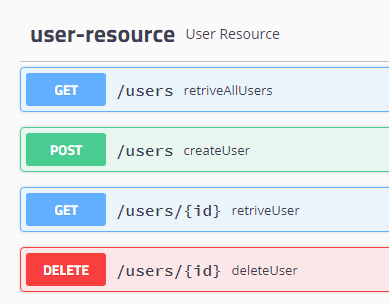


If we want to share the documentation of web service with customers, we can share this JSON file.

Now type the URI http://localhost:8080/swagger-ui.html in the browser. It shows the documentation of the services that we have created.



We can also expand the services to see which operations are present in service. In the following image, we have expanded user resource service.

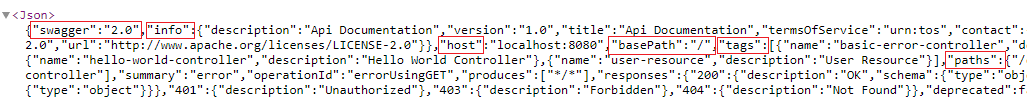


Introduction to Swagger Documentation Format

In this section, we look at the generated documentation in detail. Swagger is a specification for documenting REST API. It specifies the format (URL, method, and representation) to describe REST web services. It also provides tools to generate/compute the documentation from the application code.

As an application developer, we write web services using a framework, Swagger scans application code, and exposes the documentation on URL. A client can consume this URL and learn how to use REST web services: which HTTP methods to call on which URL, which input documents to send, which status code to expect, etc.

We're going to see what is inside the Swagger documentation. When we have a close look at the documentation of the web API, we see some important elements in the starting of the documentation, as shown in the following image.



There are following important swagger elements that are present in the Swagger documentation.

History of Java

* **swagger:** It specifies the version specification of Swagger, which we are using.
* **info:** The info tab contains the information about API like description, version of API, the title of API, termOfServices, and URL.
* **host:** It specifies the host where we are hosting the service.
* **basePath:** It is used in URI after the port number and before the API.
* **tags:** We can assign tags to our resources. It is used to group resources in multiple categories.
* **paths:** It specifies the path of resources that we are exposing and the different operations that can be performed on these resources.
* **definitions:** It includes the different elements that we have used in our API.

We will discuss three elements info, paths, and definitions in detail.

Let's see what is inside the info element:

1. "info":{"description":"Api Documentation","version":"1.0","title":"Api Documentation","termsOfService":"urn:tos","contact":{},"license":{"name":"Apache 2.0","url":"http://www.apache.org/licenses/LICENSE-2.0"}},

* **description:** It contains the high level information of API.
* **version:** It shows the version of API which we are exposing.
* **title:** It specifies the title of API.
* **termOfService:** It specifies the term of service, if any.
* **contact:** It specifies the contact detail of a person, if any.
* **license:** It specifies the default license Apache 2.0.

Let's expand the path element. It contains all the path that we are exposing.

1. paths: {
2. /error: {-}
3. /hello-world: {-}
4. /hello-world-bean: {-}
5. /hello-world-internationalized: {-}
6. /hello-world/path-variable/{name}: {-}
7. /users: {-}
8. /users/{id}: {-}
9. },

The two most important resources are "**/users**" and "**/users/{id}**". These resources exposes the group of users. Let's expand these two resources one by one.

**Expanding "/users" resource:**

1. "/users":{"get":{"tags":["user-resource"],"summary":"retriveAllUsers","operationId":"retriveAllUsersUsingGET","produces":["\*/\*"],"responses":{"200":{"description":"OK","schema":{"type":"array","items":{"$ref":"#/definitions/User"}}},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}},"deprecated":**false**},"post":{"tags":["user-resource"],"summary":"createUser","operationId":"createUserUsingPOST","consumes":["application/json"],"produces":["\*/\*"],"parameters":[{"in":"body","name":"user","description":"user","required":**true**,"schema":{"$ref":"#/definitions/User"}}],"responses":{"200":{"description":"OK","schema":{"type":"object"}},"201":{"description":"Created"},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}},"deprecated":**false**}},

The above resource contains the two operations **get** and **post** that can be performed. We can use get operation to retrieve all the users and post operation to post a user.

Inside the get operation, we get all the response status present there. Response status 200 denotes the successful creation of a user, 401 denotes the unauthorized access of resources, 404 denotes not found, and 403 denotes the forbidden. When we look at the status 200, there is a schema definition. Schema definition shows that we are sending an array of the user as a response. An array of the user is present in the definitions. Similarly, we can also expand the definitions tag to see the definition of the user.

In addition to a POST request, we have parameters that send as part of the body of the request. We accept an input type **user** as the body of the request.

**Expand "/users/{id}" resource:**

1. "/users/{id}":{"get":{"tags":["user-resource"],"summary":"retriveUser","operationId":"retriveUserUsingGET","produces":["\*/\*"],"parameters":[{"name":"id","in":"path","description":"id","required":**true**,"type":"integer","format":"int32"}],"responses":{"200":{"description":"OK","schema":{"$ref":"#/definitions/Resource«User»"}},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}},"deprecated":**false**},"delete":{"tags":["user-resource"],"summary":"deleteUser","operationId":"deleteUserUsingDELETE","produces":["\*/\*"],"parameters":[{"name":"id","in":"path","description":"id","required":**true**,"type":"integer","format":"int32"}],"responses":{"200":{"description":"OK"},"204":{"description":"No Content"},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"}},"deprecated":**false**}}},

The /users/{id} resource allows two operations **get** and **delete**. Inside the delete method, there is a parameter called id. This id we are accepting in the path while in the post request, we put content as a part of the body of the request.

Definition defines different kinds of objects that are being used. These definitions are used in the different operations exposed by each resource. When we perform get operation on /users, it returns a list of users. This resource of the user sending back to get the operation of the resource **/user/{id}** and the resource of the user contains the additional links. The definition of links is also present in the resource of user type.

Links contains a rel and href. A rel is all **-users**, and href is the link to a particular resource.

There are two ways to expose documentation to the client:

* Download the documentation from http://localhost:8080/v2/api-docs as JSON and send it to clients.
* Share the link of Swagger UI http://localhost:8080/swagger-ui.html. It is a UI that describes all the operations that are ready to expose.

Enhancing Swagger Documentation with Custom Annotations

In the previous section, we have learned about API documentation. We saw a high-level overview structure of the Swagger documentation. In this section, we will customize the Swagger element **info**. Swagger annotations are defined in the swagger-annotations-1.5.20.jar file.

**Step 1:** Open the **SwaggerConfig.java**.

**Step 2:** Create a constant **DEFAULT\_API\_INFO** of type ApiInfo.

1. **private** **static** **final** ApiInfo DEFAULT\_API\_INFO = **null**;

**Step 3:** Hold the **ctrl** key and click on the constant type (ApiInfo). The ApiInfo class will open.

Competitive questions on Structures in Hindi

Keep Watching

**Step 4:** Copy the two constants **DEFAULT \_CONTACT** and **DEFAULT** from the class. or copy the following code and paste it in the SwaggerConfig.java. Rename the constant DEFAULT to DEFAULT\_API\_INFO.

1. **public** **static** **final** Contact DEFAULT\_CONTACT = **new** Contact("", "", "");
2. **public** **static** **final** ApiInfo DEFAULT\_API\_INFO = **new** ApiInfo("Api Documentation", "Api Documentation", "1.0", "urn:tos", DEFAULT\_CONTACT, "Apache 2.0", "http://www.apache.org/licenses/LICENSE-2.0", **new** ArrayList<VendorExtension>());

**Step 5:** Configure the contact details of the developer or organization.

1. **public** **static** **final** Contact DEFAULT\_CONTACT = **new** Contact("Andrew", "http://www.javatpoint.com", "demo@javatpoint.com");

**Step 6:** Configure DEFAULT\_API\_INFO. In this configuration, provide all the information which we want to show in the Swagger documentation.

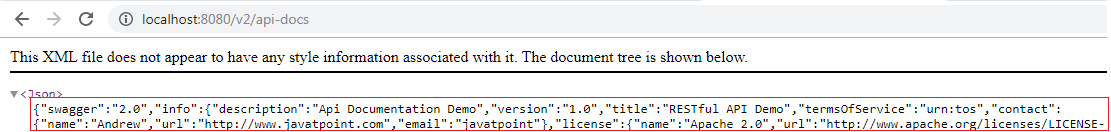
1. **public** **static** **final** ApiInfo DEFAULT\_API\_INFO = **new** ApiInfo("RESTful API Demo", "Api Documentation Demo", "1.0", "urn:tos",
2. DEFAULT\_CONTACT, "Apache 2.0", "http://www.apache.org/licenses/LICENSE-2.0", **new** ArrayList<VendorExtension>());

**SwaggerConfig.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.ArrayList;
3. **import** org.springframework.context.annotation.Bean;
4. **import** org.springframework.context.annotation.Configuration;
5. **import** springfox.documentation.service.ApiInfo;
6. **import** springfox.documentation.service.Contact;
7. **import** springfox.documentation.service.VendorExtension;
8. **import** springfox.documentation.spi.DocumentationType;
9. **import** springfox.documentation.spring.web.plugins.Docket;
10. **import** springfox.documentation.swagger2.annotations.EnableSwagger2;
11. //Configuration
12. @Configuration
13. //Enable Swagger
14. @EnableSwagger2
15. **public** **class** SwaggerConfig
16. {
17. //configuring the contact detail
18. **public** **static** **final** Contact DEFAULT\_CONTACT = **new** Contact("Andrew", "http://www.javatpoint.com", "javatpoint");
19. //configuring DEFAULT\_API\_INFO
20. **public** **static** **final** ApiInfo DEFAULT\_API\_INFO = **new** ApiInfo("RESTful API Demo", "Api Documentation Demo", "1.0", "urn:tos",
21. DEFAULT\_CONTACT, "Apache 2.0", "http://www.apache.org/licenses/LICENSE-2.0", **new** ArrayList<VendorExtension>());
22. //creating bean
23. @Bean
24. **public** Docket api()
25. {
26. ApiInfo apiInfo;
27. **return** **new** Docket(DocumentationType.SWAGGER\_2).apiInfo(DEFAULT\_API\_INFO);
28. }
29. }

**Step 7:** Restart the application.

**Step 8:** Open the browser and type the URI http://localhost:8080/v2/api-docs. It shows the updated contact detail and API info in the documentation.



Accepting and producing XML format

We should be more specific about what we produce and what we consume. So, in the next step, we will add content negotiation. We can accept input in application/json or application/xml and produce response in application/json or application/xml format.

Let's specify the content negotiation in our project.

**Step 1:** In the **SwaggerConfig.java**, goto the Docket api() and add .produces(DEFAULT\_PRODUCES\_AND\_CONSUMES).consumes(DEFAULT\_PRODUCES\_AND\_CONSUMES).

1. **return** **new** Docket(DocumentationType.SWAGGER\_2).apiInfo(DEFAULT\_API\_INFO).produces(DEFAULT\_PRODUCES\_AND\_CONSUMES).consumes(DEFAULT\_PRODUCES\_AND\_CONSUMES);

**Step 2:** Create a constant **DEFAULT\_PRODUCES\_AND\_CONSUMES**.

1. **private** **static** **final** Set<String> DEFAULT\_PRODUCES\_AND\_CONSUMES = **null**;

**Step 3:** Create a HashSet and add two values **application/json** and **application/xml**.

Note that we cannot directly pass the values to the HashSet. So we have passed a List to the constructor of HashSet.

1. **private** **static** **final** Set<String> DEFAULT\_PRODUCES\_AND\_CONSUMES = **new** HashSet<String>(Array.asList("application/json","appication/xml"));

**SwaggerConfig.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.ArrayList;
3. **import** java.util.Arrays;
4. **import** java.util.HashSet;
5. **import** java.util.Set;
6. **import** org.springframework.context.annotation.Bean;
7. **import** org.springframework.context.annotation.Configuration;
8. **import** springfox.documentation.service.ApiInfo;
9. **import** springfox.documentation.service.Contact;
10. **import** springfox.documentation.service.VendorExtension;
11. **import** springfox.documentation.spi.DocumentationType;
12. **import** springfox.documentation.spring.web.plugins.Docket;
13. **import** springfox.documentation.swagger2.annotations.EnableSwagger2;
14. //Configuration
15. @Configuration
16. //Enable Swagger
17. @EnableSwagger2
18. **public** **class** SwaggerConfig
19. {
20. //configuring the contact detail
21. **public** **static** **final** Contact DEFAULT\_CONTACT = **new** Contact("Andrew", "http://www.javatpoint.com", "javatpoint");
22. //configuring DEFAULT\_API\_INFO
23. **public** **static** **final** ApiInfo DEFAULT\_API\_INFO = **new** ApiInfo("RESTful API Demo", "Api Documentation Demo", "1.0", "urn:tos",
24. DEFAULT\_CONTACT, "Apache 2.0", "http://www.apache.org/licenses/LICENSE-2.0", **new** ArrayList<VendorExtension>());
25. //two format which we want to produce and consume
26. **private** **static** **final** Set<String> DEFAULT\_PRODUCES\_AND\_CONSUMES = **new** HashSet<String>(Arrays.asList("application/json","appication/xml"));
27. //creating bean
28. @Bean
29. **public** Docket api()
30. {
31. ApiInfo apiInfo;
32. **return** **new** Docket(DocumentationType.SWAGGER\_2).apiInfo(DEFAULT\_API\_INFO).produces(DEFAULT\_PRODUCES\_AND\_CONSUMES).consumes(DEFAULT\_PRODUCES\_AND\_CONSUMES);
33. }
34. }

**Step 4:** Open the browser and type the URI http://localhost:8080/v2/api-docs.

Enhancing Swagger Documentation with Custom Annotations

The above image shows that it consume and produces JSON and XML format.

We can add more description about user model such as date of birth must be in the past, the name must have at least five characters, etc. Let's add more description in the User model.

**Step 1:** Open the **User.java** and add **@ApiModel** annotation just above the class name. Add the description about the User model.

**@ApiModel:** It provides additional information about Swagger Models.

We have added the following description:

1. @ApiModel(description="All details about the user")

**Step 2:** Add another annotation **@ApiModelProperty** just above the **dob** variable.

**@ApiModelProperty:** It allows controlling swagger-specific definitions such as values, and additional notes.

1. @ApiModelProperty(notes="Birth date should be in the past")

**Step 3:** Similarly, add @ApiModelProperty annotation for the **name** variable.

1. @ApiModelProperty(notes="name should have atleast 5 characters")

**User.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.Date;
3. **import** javax.validation.constraints.Past;
4. **import** javax.validation.constraints.Size;
5. **import** io.swagger.annotations.ApiModel;
6. **import** io.swagger.annotations.ApiModelProperty;
7. @ApiModel(description="All details about the user")
8. **public** **class** User
9. {
10. **private** Integer id;
11. @Size(min=5, message="Name should have atleast 5 characters")
12. @ApiModelProperty(notes="name should have atleast 5 characters")
13. **private** String name;
14. @Past
15. @ApiModelProperty(notes="Birth date should be in the past")
16. **private** Date dob;
17. //default constructor
18. **protected** User()
19. {
21. }
22. **public** User(Integer id, String name, Date dob)
23. {
24. **super**();
25. **this**.id = id;
26. **this**.name = name;
27. **this**.dob = dob;
28. }
29. **public** Integer getId()
30. {
31. **return** id;
32. }
33. **public** **void** setId(Integer id)
34. {
35. **this**.id = id;
36. }
37. **public** String getName()
38. {
39. **return** name;
40. }
41. **public** **void** setName(String name)
42. {
43. **this**.name = name;
44. }
45. **public** Date getDob()
46. {
47. **return** dob;
48. }
49. **public** **void** setDob(Date dob)
50. {
51. **this**.dob = dob;
52. }
53. @Override
54. **public** String toString()
55. {
56. //return "User [id=" + id + ", name=" + name + ", dob=" + dob + "]";
57. **return** String.format("User [id=%s, name=%s, dob=%s]", id, name, dob);
58. }
59. }

**Step 4:** Restart the application.

**Step 5:** Open the browser and type the URI http://localhost:8080/v2/api-docs. If we look at the description of the User model, the description which we have specified appears here.

Enhancing Swagger Documentation with Custom Annotations

The API documentation is much important as API. The consumer of our service should not have any question about how to consume service, what are the different details, how the output looks like, etc. Everything should be clear to the consumer so that the user can understand easily.

Hence, the Swagger documentation is beneficial for the consumer of service. Swagger provides a lot of annotations that can be used to enhance model, operations, and swagger configurations.

# Monitoring APIs with Spring Boot Actuator

## Spring Boot Actuator

Spring Boot provides the actuator to monitor and manage applications effectively. It is a tool that has HTTP endpoints (the place where the resource lives). It is a sub-project of Spring Boot. It adds several production grade services to our application with less effort.

If the performance of a service goes down or failure, we should know the reason as quickly as possible. We need to build monitoring around the API, especially when we build microservices. Spring Boot has great support to provide monitoring.

To use the production-ready features, we will add **spring-boot-actuator** dependency in pom.xml.

Let's add monitoring service in our Spring Boot project.

33M

569

Hello Java Program for Beginners

**Step 1:** Open **pom.xml** and add the following dependencies:

**Spring Boot Starter Actuator:** It provides a lot of monitoring facilities around your services.

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-actuator</artifactId>
4. </dependency>

**Spring Data Rest HAL Browser:** HAL makes our API exportable, and its documentation easily discoverable from within the API itself.

1. <dependency>
2. <groupId>org.springframework.data</groupId>
3. <artifactId>spring-data-rest-hal-browser</artifactId>
4. </dependency>

Hypertext Application Language (HAL) is a simple language that gives a consistent and easy way to hyperlink between resources in API. The spring boot starter actuator is actually in the HAL format. HAL browser searches for APIs and identifies the links. It shows the link on the screen so that we can easily browse through the API.

**Step 2:** Restart the application.

**Start 3:** Type the URL **localhost:8080/actuator** in the browser and hit the enter key. If it does not work, use the URL **localhost:8080/application**.

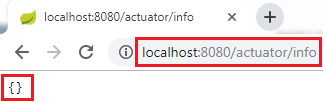
It launches the actuator that shows the three URLs: self, health, and info.

1. {"\_links":{"self":{"href":"http://localhost:8080/actuator","templated":**false**},"health":{"href":"http://localhost:8080/actuator/health","templated":**false**},"health-component":{"href":"http://localhost:8080/actuator/health/{component}","templated":**true**},"health-component-instance":{"href":"http://localhost:8080/actuator/health/{component}/{instance}","templated":**true**},"info":{"href":"http://localhost:8080/actuator/info","templated":**false**}}}

When we click on the **health** URL, it shows the health of the application. In the following image, status **up** denotes that application is running.



When we click on the **info** URL, it shows the information of the application. A pair of blank curly braces denote that there is no information is available.



To enable the information, we need to configure the property.

* Open **application.properties** file and enable **web exposure**.

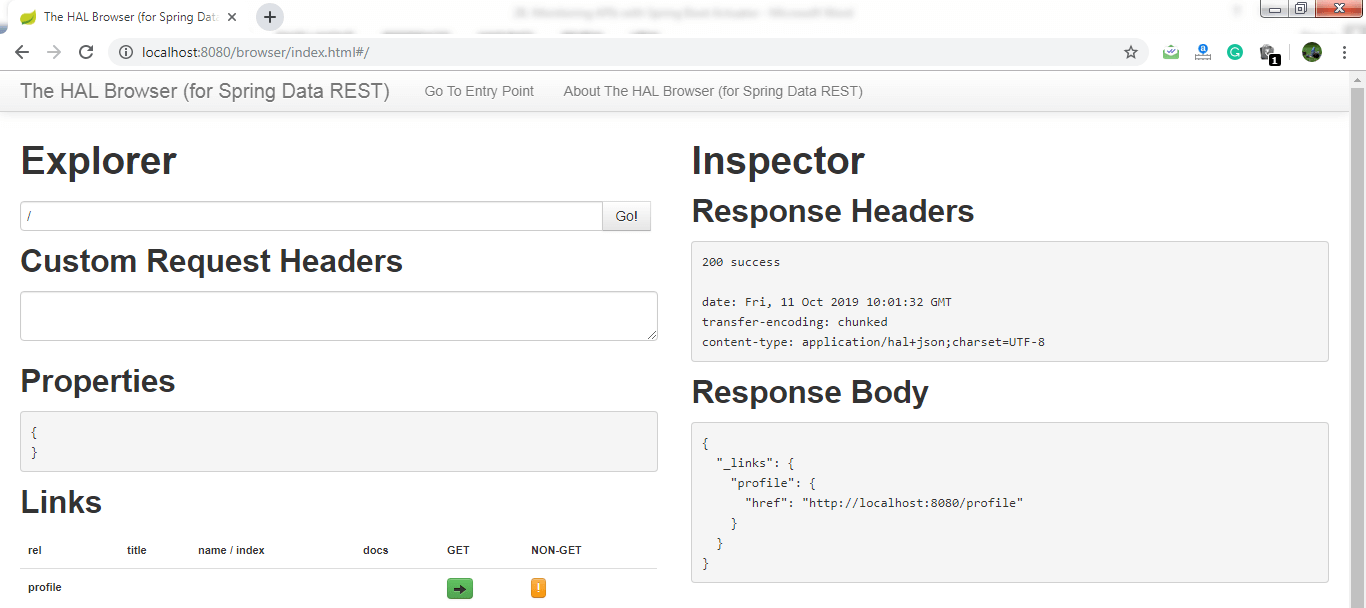
1. management.endpoints.web.exposure.include=\*

* Restart the application.
* Restart the actuator by using the URL **localhost:8080/actuator**.

It shows a lot of URLs.

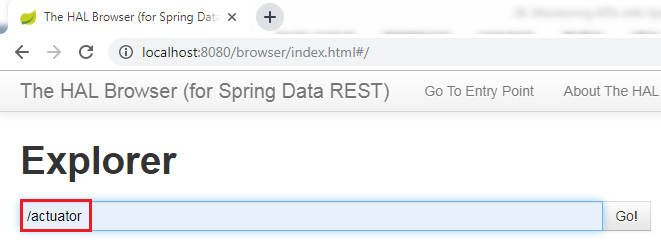
## The HAL Browser

To access the HAL browser, type **localhost:8080** in the browser and hit the enter key.



Now we can access the actuator through the HAL browser.

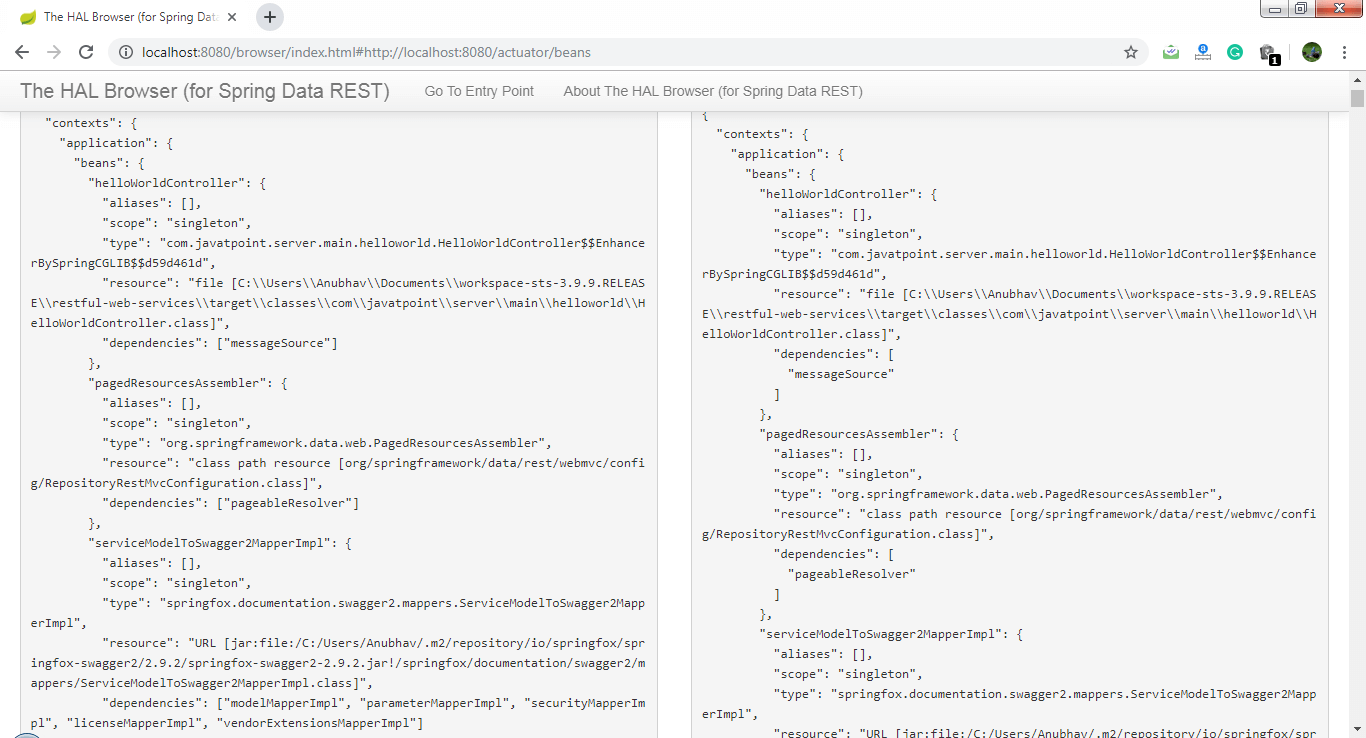
Type **/actuator** in the Explorer's text box and click on the Go button.



It shows all the things related to the actuator. The most important thing in the actuator is **beans**.



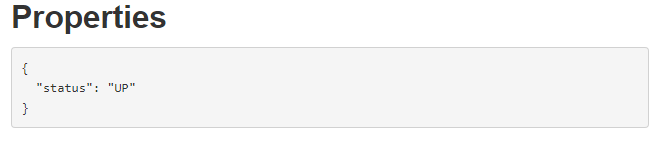
When we click on the bean's arrow, it shows all the beans configured in the spring boot project.



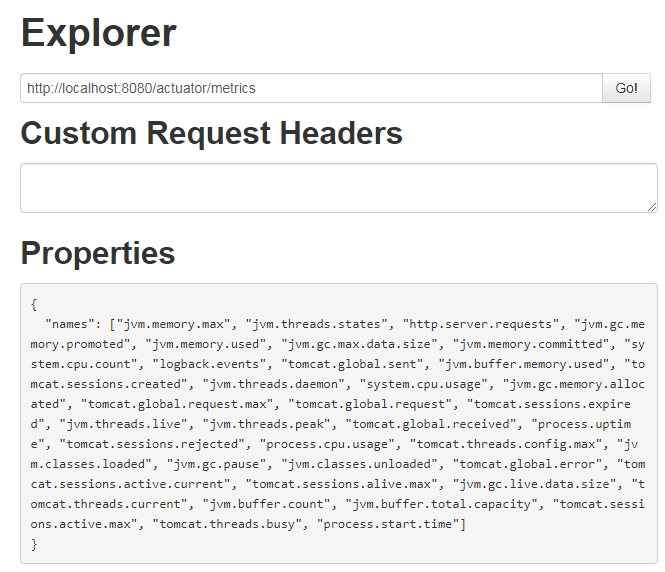
If we want to check the status of the application, we can click on the health link.

Monitoring APIs with Spring Boot Actuator

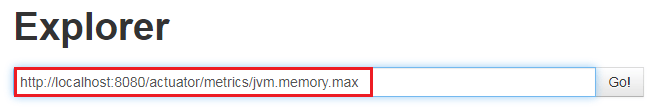
It shows the health of the application.



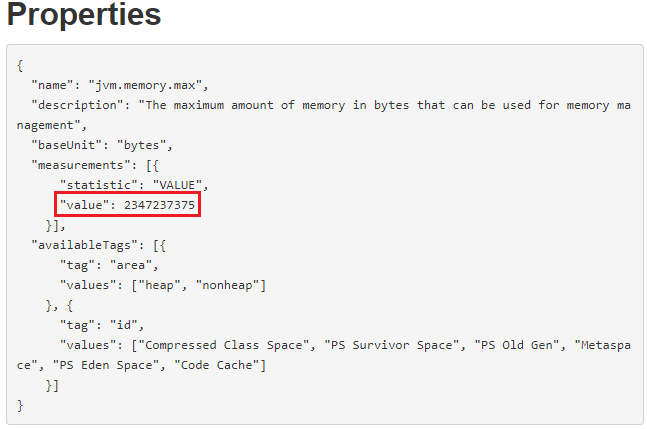
There is another link in the actuator named **metrics**. It shows the list of valid metrics.



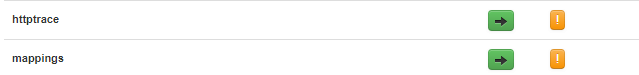
Suppose we want to know how much memory is used by the application. We have accessed **/jvm.memory.max**.



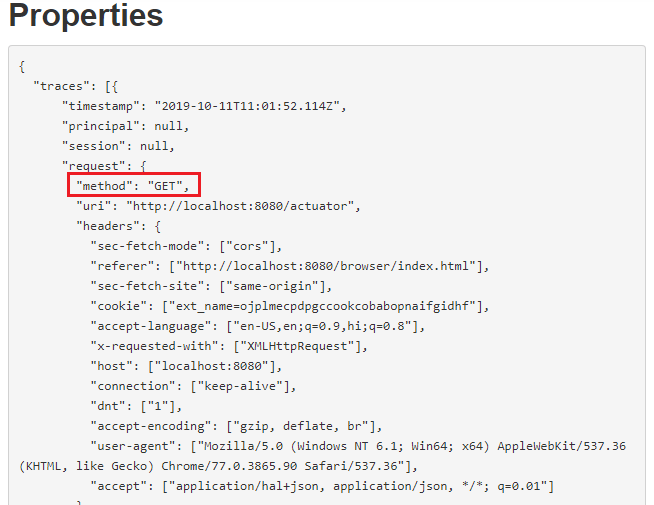
In the following image, the **value** indicates the **maximum** memory used by the application.



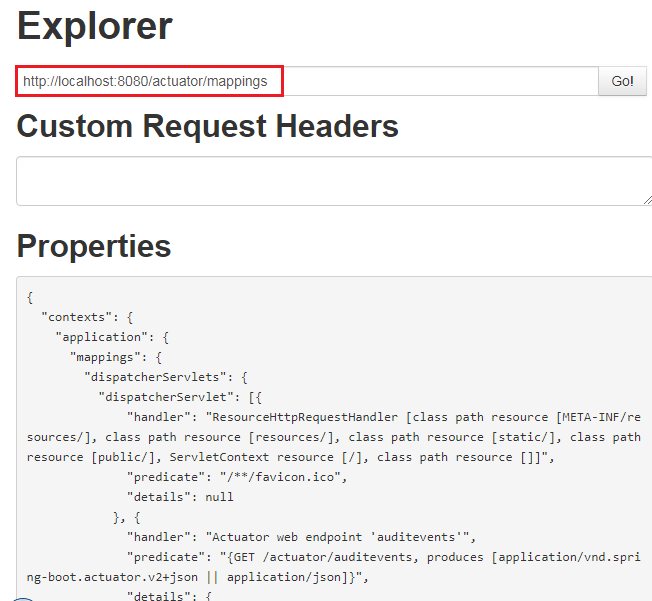
There are two important links **httptrace** and **mapping** that are present in the actuator.



The httptrace shows all the requests that we have executed earlier. We can see all the details of previously executed requests, as shown below:



The mapping shows all the different things that are mapped to URLs. Whenever we are creating web services or web applications, we are mapping a lot of URLs.



In this section, we have learned what we can do with the actuator.

Implementing Static Filtering for RESTful Services

In this section, we will learn how to filter the request.

Filters are one of the important features provided by the JAX-RS framework. It is used in various contexts. It may be applied on either request to a resource or the response from a resource, or both.

Consider a scenario in which we do not want to show some class members in the response. This process is called filtering. Jackson has two annotations that are used in filtering are: **@JsonIgnore** and **@JsonIgnoreProperties**.

@JsonIgnore

It is a member or method level annotation. It expects that the properties to be excluded are marked one by one. If we want to eliminate a member from the process of serialization and deserialization, we can annotate the actual property or its setter or getter.

Java Try Catch

Let's create a filter that filters the response. We will not touch the user example, but instead of this, we will create a new controller and bean to perform filtering.

**Step 1:** Create a Controller class with the name FilteringController.java in the package **com.javatpoint.server.main.filtering**.

**Step 2:** Create a bean with the name **SomeBean**.

**FilteringController.java**

1. **package** com.javatpoint.server.main.filtering;
2. **import** org.springframework.web.bind.annotation.RequestMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. @RestController
5. **public** **class** FilteringController
6. {
7. @RequestMapping("/filtering")
8. **public** SomeBean retrieveSomeBean()
9. {
10. **return** **new** SomeBean("Amit", "9999999999","39000");
11. }
12. }

**Step 3:** Create a class with the name **SomeBean.java**. Define three attributes **name, phone,** and **salary**.

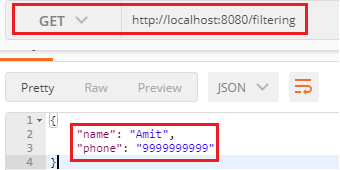
**Step 4:** Generate Constructors.

**Step 5:** Generate Getters and Setters.

**SomeBean.java**

1. **package** com.javatpoint.server.main.filtering;
2. **import** com.fasterxml.jackson.annotation.JsonIgnore;
3. **public** **class** SomeBean
4. {
5. **private**  String name;
6. **private**  String phone;
7. //JsonIgnore indicates that the annotated method or field is to be ignored
8. @JsonIgnore
9. **private**  String salary;
10. //generating constructor
11. **public** SomeBean(String name, String phone, String salary)
12. {
13. **super**();
14. **this**.name = name;
15. **this**.phone = phone;
16. **this**.salary = salary;
17. }
18. **public** String getName()
19. {
20. **return** name;
21. }
22. **public** **void** setName(String name)
23. {
24. **this**.name = name;
25. }
26. **public** String getPhone()
27. {
28. **return** phone;
29. }
30. **public** **void** setPhone(String phone)
31. {
32. **this**.phone = phone;
33. }
34. **public** String getSalary()
35. {
36. **return** salary;
37. }
38. **public** **void** setSalary(String salary)
39. {
40. **this**.salary = salary;
41. }
42. }

**Step 6:** Open the REST client **Postman** and send the GET request. It returns two fields: name and phone. The field salary will not send with the response.



The field salary will not be sent in the response whether we are sending one SomeBean as a response, or a list of SomeBeans as a response.

**Step 7:** Create another bean that returns a list of SomeBean.

**FilteringController.java**

1. **package** com.javatpoint.server.main.filtering;
2. **import** java.util.Arrays;
3. **import** java.util.List;
4. **import** org.springframework.web.bind.annotation.RequestMapping;
5. **import** org.springframework.web.bind.annotation.RestController;
6. @RestController
7. **public** **class** FilteringController
8. {
9. //returning a single bean as response
10. @RequestMapping("/filtering")
11. **public** SomeBean retrieveSomeBean()
12. {
13. **return** **new** SomeBean("Amit", "9999999999","39000");
14. }
15. //returning a list of SomeBeans as response
16. @RequestMapping("/filtering-list")
17. **public** List<SomeBean> retrieveListOfSomeBeans()
18. {
19. **return** Arrays.asList(**new** SomeBean("Saurabh", "8888888888","20000"), **new** SomeBean("Devesh", "1111111111","34000"));
20. }
21. }

**Step 8:** Send a GET request again with the URI http://localhost/filtering-list that returns a list of SomeBeans.



There is another approach to use an annotation **@JsonIgnoreProperties**.

@JsonIgnoreProperties

**@JsonIgnoreProperties** is a class-level annotation. It ignores the logical properties in JSON serialization and deserialization.

In the following **SomeBean.java** file, we have specified the property name and phone, which we want to ignore in the response. These two properties will take part in the JSON serialization and deserialization.

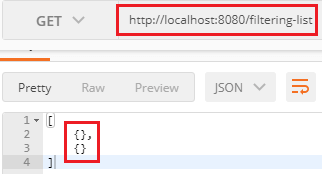
If the property salary is annotated with @JsonIgnore, then all the properties ignored in the JSON serialization and deserialization.

In other words, the union of logical properties ignored by @JsonIgnore and @JsonIgnoreProperties annotation is considered to be ignored in JSON serialization and deserialization.

**SomeBean.java**

1. **package** com.javatpoint.server.main.filtering;
2. **import** com.fasterxml.jackson.annotation.JsonIgnore;
3. **import** com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4. @JsonIgnoreProperties({"name", "phone"})
5. **public** **class** SomeBean
6. {
7. **private**  String name;
8. **private**  String phone;
9. //JsonIgnore indicates that the annotated method or field is to be ignored
10. @JsonIgnore
11. **private**  String salary;
12. //generating constructor
13. **public** SomeBean(String name, String phone, String salary)
14. {
15. **super**();
16. **this**.name = name;
17. **this**.phone = phone;
18. **this**.salary = salary;
19. }
20. **public** String getName()
21. {
22. **return** name;
23. }
24. **public** **void** setName(String name)
25. {
26. **this**.name = name;
27. }
28. **public** String getPhone()
29. {
30. **return** phone;
31. }
32. **public** **void** setPhone(String phone)
33. {
34. **this**.phone = phone;
35. }
36. **public** String getSalary()
37. {
38. **return** salary;
39. }
40. **public** **void** setSalary(String salary)
41. {
42. **this**.salary = salary;
43. }
44. }

When we fire a GET request, it returns an empty list because the properties name and phone are specified in the @JsonIgnoreProperties and the property salary annotated with @JsonIgnore. Hence it returns the empty list.



Now remove the annotation **@JsonIgnore** and again fire a GET request. It returns only the salary property.



Whatever we have done is called static filtering. Suppose we want to ignore the name in one scenario and the salary in another scenario, we cannot do that using the static filtering. To implement this type of filtering, we use dynamic filtering.

Implementing Dynamic Filtering for RESTful Services

In the previous section, we have performed static filtering. Now we are moving to dynamic filtering.

In the dynamic filtering, we define **different filters for different services**, according to need. So there exists the concept of dynamic filtering.

Suppose there are three fields: name, phone, and salary. We want to send two fields: name and salary for the first service and name and phone for the second service.

But there is a limitation in dynamic filtering. We cannot configure the **dynamic filtering** directly in the bean. We need to start configuring filtering, where we are retrieving the values. To implement dynamic filtering, we use a class named MappingJacksonValue. If we look at the class definition, we found the filter method definition there.

Difference between JDK, JRE, and JVM

Let's see how to implement dynamic filtering in our project.

In the following example, we will send a name and salary for "**/filtering**" mapping.

**Step 1:** Open **FilteringController.java** file.

**Step 2:** Create a constructor of MappingJacksonValue class and pass a bean (someBean) as a constructor argument. We want to create a mapping Jackson value for this particular bean.

1. MappingJacksonValue mapping = **new** MappingJacksonValue (someBean);

**Step 3:** To configure the filters, we need to create them. To create a filter, declare local variable filters of type **FilterProvider**. FilterProvider is an abstract class. It has a single implementation of the **SingleFilterProvider** method. Invoke the addFilter() method that has two parameters String id and SimpleBeanPropretyFilter filter.

1. FilterProvider filters=**new** SimpleFilterProvider().addFilter("SomeBeanFilter", filter);

**Step 4:** Invoke the static method **filterOutAllExcept()** of the class SimpleBeanPropertyFilter class. It filters all the fields in response except the fields which we have specified. We want to send the name and salary field in the response, so we have specified these two fields.

1. SimpleBeanPropertyFilter filter=SimpleBeanPropertyFilter.filterOutAllExcept("name", "salary");

**Step 5:** Configure the filters.

1. mapping.setFilters(filters);

**Step 6:** Instead of returning the someBean return **mapping**.

1. **return** mapping;

**Step 7:** We have returned mapping, so we are required to change the return type of the method to MappingJacksonValue.

1. **public** MappingJacksonValue retrieveSomeBean()
3. @RequestMapping("/filtering")
4. **public** MappingJacksonValue retrieveSomeBean()
5. {
6. SomeBean someBean=**new** SomeBean("Amit", "9999999999","39000");
7. //invoking static method filterOutAllExcept()
8. SimpleBeanPropertyFilter filter=SimpleBeanPropertyFilter.filterOutAllExcept("name", "salary");
9. //creating filter using FilterProvider class
10. FilterProvider filters=**new** SimpleFilterProvider().addFilter("SomeBeanFilter",filter);
11. //constructor of MappingJacksonValue class  that has bean as constructor argument
12. MappingJacksonValue mapping = **new** MappingJacksonValue(someBean);
13. //configuring filters
14. mapping.setFilters(filters);
15. **return** mapping;
16. }

**Remember:** The list of valid filters should be defined in the bean. If we don't do that it will return all the fields.

**Step 8:** Open **SomeBean.java** file and define a filter by using the annotation **@JsonFilter**. It is used at class level. It defines a filter name which we filter out properties in JSON serialization.

1. @JsonFilter("SomeBeanFilter")

**SomeBean.java**

1. **package** com.javatpoint.server.main.filtering;
2. **import** com.fasterxml.jackson.annotation.JsonFilter;
3. @JsonFilter("SomeBeanFilter")
4. **public** **class** SomeBean
5. {
6. **private**  String name;
7. **private**  String phone;
8. //JsonIgnore indicates that the annotated method or field is to be ignored
9. //@JsonIgnore
10. **private**  String salary;
11. //generating constructor
12. **public** SomeBean(String name, String phone, String salary)
13. {
14. **super**();
15. **this**.name = name;
16. **this**.phone = phone;
17. **this**.salary = salary;
18. }
19. **public** String getName()
20. {
21. **return** name;
22. }
23. **public** **void** setName(String name)
24. {
25. **this**.name = name;
26. }
27. **public** String getPhone()
28. {
29. **return** phone;
30. }
31. **public** **void** setPhone(String phone)
32. {
33. **this**.phone = phone;
34. }
35. **public** String getSalary()
36. {
37. **return** salary;
38. }
39. **public** **void** setSalary(String salary)
40. {
41. **this**.salary = salary;
42. }
43. }

Let's implement the dynamic filtering in the second method. In this method we will return name and phone field for "**/filtering-list**" mapping.

**Step 1:** In this method, we first change the return type of the method to MappingJacksonValue.

**Step 2:** Create a list of SomeBean.

1. List<SomeBean> list=Arrays.asList(**new** SomeBean("Saurabh", "8888888888","20000"), **new** SomeBean("Devesh", "1111111111","34000"));

**Step 3:** Specify the field name which we want to send in the response. In our case, we have specified **name** and **phone**.

1. SimpleBeanPropertyFilter filter=SimpleBeanPropertyFilter.filterOutAllExcept("name", "phone");

**Step 4:** Pass the list in **MappingJacksonValue** constructor.

**Step 5:** Return mapping.

1. @RequestMapping("/filtering-list")
2. **public** MappingJacksonValue retrieveListOfSomeBeans()
3. {
4. List<SomeBean> list=Arrays.asList(**new** SomeBean("Saurabh", "8888888888","20000"), **new** SomeBean("Devesh", "1111111111","34000"));
5. //invoking static method filterOutAllExcept()
6. SimpleBeanPropertyFilter filter=SimpleBeanPropertyFilter.filterOutAllExcept("name", "phone");
7. FilterProvider filters=**new** SimpleFilterProvider().addFilter("SomeBeanFilter",filter);
8. //constructor of MappingJacksonValue class that has list as constructor argument
9. MappingJacksonValue mapping = **new** MappingJacksonValue(list);
10. //configuring filter
11. mapping.setFilters(filters);
12. **return** mapping;
13. }

**FilteringController.java**

1. **package** com.javatpoint.server.main.filtering;
2. **import** java.util.Arrays;
3. **import** java.util.List;
4. **import** org.springframework.http.converter.json.MappingJacksonValue;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
6. **import** org.springframework.web.bind.annotation.RestController;
7. **import** com.fasterxml.jackson.databind.ser.impl.SimpleBeanPropertyFilter;
8. **import** com.fasterxml.jackson.databind.ser.impl.SimpleFilterProvider;
9. **import** com.fasterxml.jackson.databind.ser.FilterProvider;
10. @RestController
11. **public** **class** FilteringController
12. {
13. //returning a single bean as response
14. //values to send name and salary
15. @RequestMapping("/filtering")
16. **public** MappingJacksonValue retrieveSomeBean()
17. {
18. SomeBean someBean=**new** SomeBean("Amit", "9999999999","39000");
19. //invoking static method filterOutAllExcept()
20. SimpleBeanPropertyFilter filter=SimpleBeanPropertyFilter.filterOutAllExcept("name", "salary");
21. //creating filter using FilterProvider class
22. FilterProvider filters=**new** SimpleFilterProvider().addFilter("SomeBeanFilter",filter);
23. //constructor of MappingJacksonValue class  that has bean as constructor argument
24. MappingJacksonValue mapping = **new** MappingJacksonValue(someBean);
25. //configuring filters
26. mapping.setFilters(filters);
27. **return** mapping;
28. }
29. //returning a list of SomeBeans as response
30. //values to send name and phone
31. @RequestMapping("/filtering-list")
32. **public** MappingJacksonValue retrieveListOfSomeBeans()
33. {
34. List<SomeBean> list=Arrays.asList(**new** SomeBean("Saurabh", "8888888888","20000"), **new** SomeBean("Devesh", "1111111111","34000"));
35. //invoking static method filterOutAllExcept()
36. SimpleBeanPropertyFilter filter=SimpleBeanPropertyFilter.filterOutAllExcept("name", "phone");
37. FilterProvider filters=**new** SimpleFilterProvider().addFilter("SomeBeanFilter",filter);
38. //constructor of MappingJacksonValue class that has list as constructor argument
39. MappingJacksonValue mapping = **new** MappingJacksonValue(list);
40. //configuring filter
41. mapping.setFilters(filters);
42. **return** mapping;
43. }
44. }

Now Open the REST Client **Postman** and send a **GET** request with the URI http://localhost:8080/filtering. It returns the two fields name and salary in the response as shown in the following image.



Again, send a GET request with the URI http://localhost:8080/filtering-list. It returns a list with two fields name and phone in the response, as shown in the following image.



# Versioning RESTful Web Services-Basic Approach With URIs

Versioning is the most important and difficult part of the API as it takes backward API compatible. Versioning helps us to iterate faster when the changes are identified. We should always version our Web API.

Consider a scenario in which we have a Web API that is up (status) and running. The users are consuming that API. Now we want to add more functionality in the Web API but want to keep the existing functionality unchanged. There may be few users who still want to use the old API while the other users want a new version of API with new or extended features. It is the scenario where Web API versioning comes into existence.

## When we required versioning:

When we made a **breaking change** in Web API, we should up versioned the API. Breaking changes includes:

* A change in the format of the response data for one or more calls.
* Change in the response type.
* Remove any part of the API.

Breaking changes should always result in a change to the major version number for an API or content response type.

32.5M

629

Difference between JDK, JRE, and JVM

Non-breaking changes (adding new points or new response parameters) do not require a change to the major version number. However, it can be helpful to track the minor version of the APIs.

## How to Version

The most commonly used approaches fall into three categories:

* URI Versioning
* Versioning using Custom Request Header
* Versioning using Accept Header

### **URI Versioning**

URI versioning is the most straightforward approach. It specified in the URL as a query string. It violates the principle that a URI should refer to a unique resource. You are also guaranteed to break client integration when a version is updated. **Twitter** uses URI versioning.

**Example**

<http://api.demo.com/v1>  
[http://apiv1.demo.com](http://apiv1.demo.com/)

The version need not be numeric, nor specified using v[x] syntax. Alternatives include the date, project name, season, or other identifiers that are meaningful enough to change as the version change.

### **Versioning using Custom Request Header**

A custom header allows us to preserve our URLs. It is a duplicate of the content negotiation behavior implemented by the existing Accept header. Version information is specified in the Request Header without the need for any change in the URL. **Microsoft** uses the request header versioning. The user cannot access request header versioning in the normal browser (chrome). We are required a special plugin to access them on the browser.

**Example**

Accept-version: v1  
Accept-version: v2

### **Versioning using Accept Header**

Accept header define the media type and character encodings. We can also pass version information for Web API through accept headers without changing the URL. It is also known as media type versioning or content negotiation or accept header. Github uses the accept header versioning. The user cannot access accept-header versioning in the normal browser (chrome). We are required a special plugin to access them on the browser.

**Example**

Accept: application/vnd.demo.v1+json Accept:application/vnd.demo+json;version=1.0

Let's see how to implement versioning in the project.

### **URI Versioning**

**Step 1:** Create a class with the name **PersonV1.java** in the package **com.javatpoint.server.main.versioning**. PersonV1 denotes the first version of API. The initial version of API has a name variable.

**PersonV1.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** PersonV1
3. {
4. **private** String name;
5. }

**Step 2:** Over a period, we recognize the need for having the first name and last name separately. So we created a class with the name **Person2.java**. It denotes the second version of API.

**PersonV2.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** PersonV2
3. {
4. **private** Name name;
5. }

**Step 3:** Create a class with the name **Name.java** that has two variables **firstName** and lastName separately.

**Name.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** Name
3. {
4. **private** String firstName;
5. **private** String lastName;
6. }

The old version is still returning the full name, and the second version is returning firstName and lastName separately. Now we are required to create two different versions of the same service.

Let's see how to create two different versions of the same service and what are the different versions of the service are present.

**Step 4:** In the **Name.java** file, Generate Getters and Setters, Generate Constructor using Fields. Create a no-argument constructor of the class Name.

**Name.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** Name
3. {
4. **private** String firstName;
5. **private** String lastName;
6. //no argument constructor
7. **public** Name()
8. {
10. }
11. **public** Name(String firstName, String lastName)
12. {
13. **super**();
14. **this**.firstName = firstName;
15. **this**.lastName = lastName;
16. }
17. **public** String getFirstName()
18. {
19. **return** firstName;
20. }
21. **public** **void** setFirstName(String firstName)
22. {
23. **this**.firstName = firstName;
24. }
25. **public** String getLastName()
26. {
27. **return** lastName;
28. }
29. **public** **void** setLastName(String lastName)
30. {
31. **this**.lastName = lastName;
32. }
33. }

**Step 5:** Open **PersonV1.java** class. Generate Getters and Setters, Generate Constructor using Fields. Create a no argument constructor of the class PersonV1.java.

**PersonV1.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** PersonV1
3. {
4. **private** String name;
5. //no argument constructor
6. **public** PersonV1()
7. {
8. **super**();
9. }
10. **public** PersonV1(String name)
11. {
12. **super**();
13. **this**.name = name;
14. }
15. **public** String getName()
16. {
17. **return** name;
18. }
19. **public** **void** setName(String name)
20. {
21. **this**.name = name;
22. }
23. }

**Step 6:** Open PersonV2.java. Generate Getters and Setters, Generate Constructor using Fields. Create a no argument constructor of the class PersonV2.java.

**PersonV2.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** PersonV2
3. {
4. **private** Name name;
5. **public** PersonV2()
6. {
7. **super**();
8. }
9. **public** PersonV2(Name name)
10. {
11. **super**();
12. **this**.name = name;
13. }
14. **public** Name getName()
15. {
16. **return** name;
17. }
18. **public** **void** setName(Name name)
19. {
20. **this**.name = name;
21. }
22. }

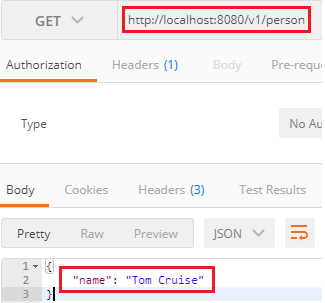
Now we need to create a service.

**Step 7:** Create a class with the name **PersonVersioningController.java**. Create two methods for different versions and map them to different URIs.

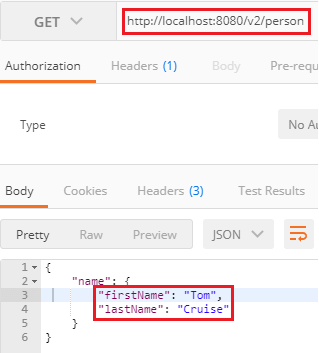
**PersonVersioningController.java**

1. **package** com.javatpoint.server.main.versioning;
2. **import** org.springframework.web.bind.annotation.RestController;
3. **import** org.springframework.web.bind.annotation.GetMapping;
4. @RestController
5. **public** **class** PersonVersoningController
6. {
7. //this method is for the first version that returns the entire name
8. @GetMapping("v1/person")
9. **public** PersonV1 personv1()
10. {
11. **return** **new** PersonV1("Tom Cruise");
12. }
13. //this method is for the second version that returns firstName and lastName separately
14. @GetMapping("v2/person")
15. **public** PersonV2 personv2()
16. {
17. **return** **new** PersonV2(**new** Name("Tom", "Cruise"));
18. }
19. }

**Step 8:** Open the **Postman** and send a **GET** request with the URI http://localhost:8080/v1/person. It returns the full name, as shown in the following image.



Change the URI http://localhost:8080/v2/person for the second version. It returns the firstName and lastName separately, as shown in the following image.



## Versioning using Request Parameter

Another way to implement versioning is by using the request parameter. Amazon uses the request parameter versioning. Open the PersonVersioningController.java and do the following changes:

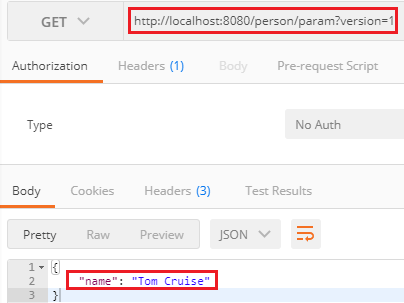
* Change the URI for the first method from /v1/person to /person/param.
* Change the name of the method from personV1 to paramV1.
* Similarly, change the URI for the second method from /v2/person to /person/param.

Both the methods have the same get mapping, so we will distinguish them by using the value and params attribute. The value attribute contains the URI, which we want to use, and the params attribute contains the parameter which distinguishes between versions.

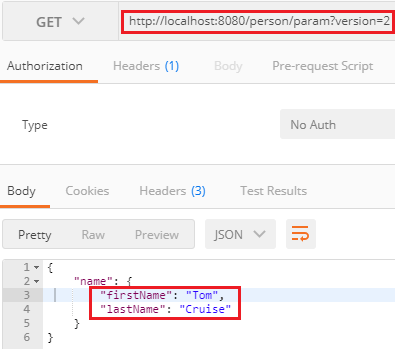
**PersonVersoningController.java**

1. **package** com.javatpoint.server.main.versioning;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. @RestController
5. **public** **class** PersonVersoningController
6. {
7. //this method is for first version that returns the entire name
8. @GetMapping(value="/person/param", params="version=1")
9. **public** PersonV1 personV1()
10. {
11. **return** **new** PersonV1("Tom Cruise");
12. }
13. //this method is for second version that returns firstName and lastName separately
14. @GetMapping(value="/person/param", params="version=2")
15. **public** PersonV2 personV2()
16. {
17. **return**  **new** PersonV2(**new** Name("Tom", "Cruise"));
18. }
19. }

Now, move to Postman and send a GET request with the URI http://localhost:8080/person/param?version=1. It returns the full name, as shown in the following image.



Again, generate a GET request with the URI http://localhost:8080/person/param?version=2 to access the second version. It returns the firstName and lastName separately, as shown in the following image.



### **Versioning using Request Header**

There is another option to do versioning using the Request Header. It is similar to content negotiation. In this method, we differentiate service based on the Request Header.

In the **PersonVersioningController.java**, do the following:

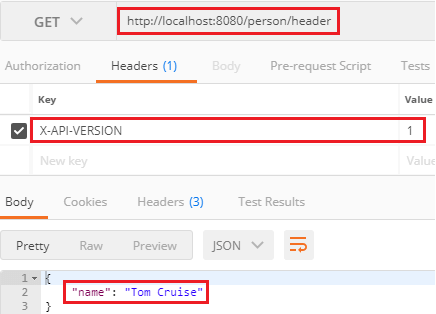
* Copy both the methods and paste in the same file.
* Change the method name paramV1 to headerV1 and paramV2 to headerV2.
* Replace the URI /person/param with /person/header and params with headers.

1. /\*---------------using request header--------------\*/
2. //this method is for first version that returns the entire name
3. @GetMapping(value="/person/header", headers="X-API-Version=1")
4. **public** PersonV1 headerV1()
5. {
6. **return** **new** PersonV1("Tom Cruise");
7. }
8. //this method is for second version that returns firstName and lastName separately
9. @GetMapping(value="/person/header", headers="X-API-Version=2")
10. **public** PersonV2 headerV2()
11. {
12. **return** **new** PersonV2(**new** Name("Tom", "Cruise"));
13. }

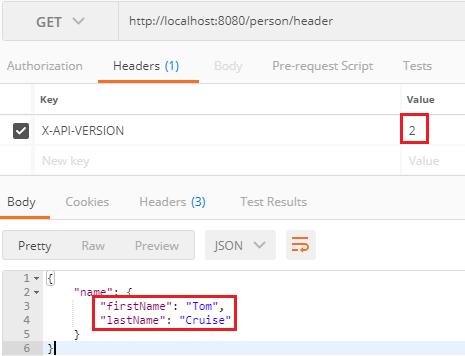
Open the **Postman**:

* Select the Headers tab and set key: X-API-VERSION and Value: 1.
* Type the URI http://localhost:8080/person/header and send a GET request.

It returns the name full name.



Let's send a GET request for version 2. For this, we need to change the value from 1 to 2 under the Headers tab. It returns the firstName and lastName separately.



## Versioning using Accept Header

Another method that is used in versioning is the Accept Header. It is also known as Content Negotiation or Accept Versioning. In this method, we use an attribute called produce. It denotes what kind of output we are generating for the specific service.

In the **PersonVersioningController.java** file, do the following:

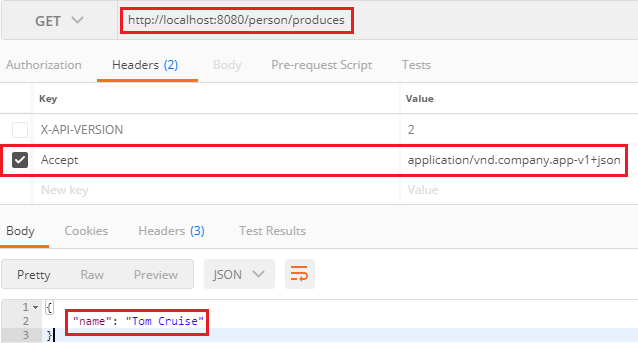
* Copy both the methods and paste in the same file.
* Change the methods name headerV1 to producesV1 and headerV2 to ProducesV2.
* Replace the URI /person/header with /person/produces and header with produces.

1. /\*---------------using accept header--------------\*/
2. //this method is for first version that returns the entire name
3. @GetMapping(value="/person/produces", produces="application/vnd.company.app-v1+json")
4. **public** PersonV1 producesV1()
5. {
6. **return** **new** PersonV1("Tom Cruise");
7. }
8. //this method is for second version that returns firstName and lastName separately
9. @GetMapping(value="/person/produces", produces="application/vnd.company.app-v2+json")
10. **public** PersonV2 producesV2()
11. {
12. **return**  **new** PersonV2(**new** Name("Tom", "Cruise"));
13. }

Open the **Postman**:

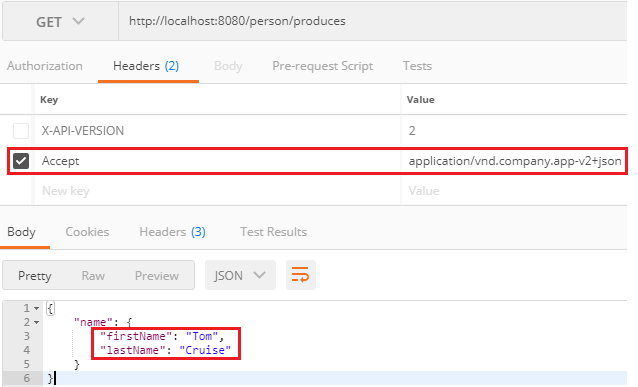
* Select the Headers tab and set key: Accept and Value: application/vnd.company.app-v1+json.
* Uncheck the X-API-VERSION key.
* Type the URI http://localhost:8080/person/produces and send a GET request.

It returns the full name.



Let's send a GET request for version 2. For this we need to change the value from **Value: application/vnd.company.app-v1+json** to **Value: application/vnd.company.app-v2+json**.

It returns the firstName and lastName separately.



Implementing Basic Authentication with Spring Security

In the previous steps, we have created some resources, but none of them is secure yet. There is no user id and password to access the resources. In this section, we will implement basic authentication.

There are multiple ways to authenticate our RESTful web services. The basic way is to use basic authentication. In the basic authentication, we send a username and password as part of our request. When we provide a username and password, it allows us to access the resource.

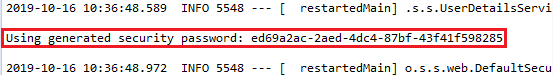
There are other advanced forms of authentication like **digest authentication**, where the password digest is created, and the digest is sent across. It does not send the actual password to the server. The other advanced form of authentication is OAuth (Open Authorization) or OAuth2 authentication.

Let's see how to implement basic authentication in web services.

**Step 1:** Open **pom.xml** and add the **spring-boot-starter-security**. It automatically configures the basic security for us.

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-security</artifactId>
4. </dependency>

**Step 2:** Restart the server, we get a **password** in the log. Each time the server starts up the password will be different.

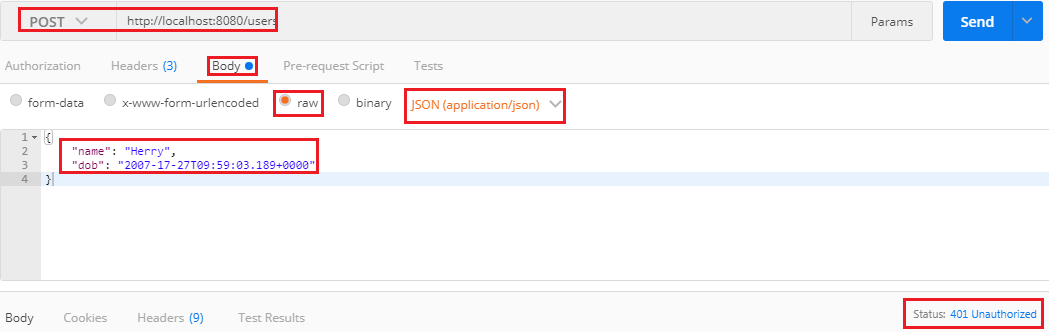


**Step 3:** Copy the password from the log.

**Step 4:** Open the REST Client **Postman** and send a POST request. We are sending a POST to create a user.

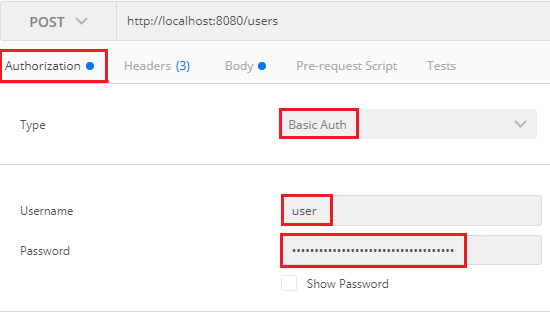
* Provide URI http://localhost:8080/users.
* Click on the Body tab and select the raw radio button.
* Select the media type JSON (application/json).
* Provide name and dob.
* Click on the Send button.

It returns the **Status: 401 Unauthorized**.



**Step 5:** In the REST client **Postman**, click on the **Authorization** tab and do the following:

* Select the type of authentication Basic Auth.
* Provide the Username. The default username is user.
* Paste the password, which we have copied from the log.
* Click on the Send button.

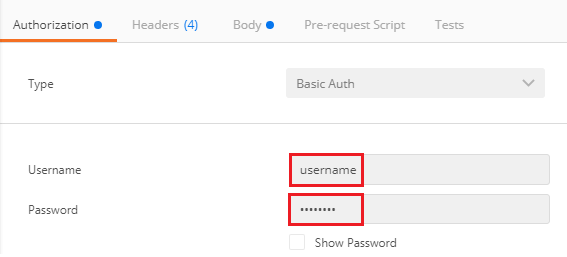


It shows the **Status: 201 Created**. There is a disadvantage that when we restart the server, the password changes again and again. The solution to this problem is that configure the username and password in the application.properties file.

**application.properties**

1. spring.security.user.name=user
2. spring.security.user.password=password

Now, move to Postman and try to send a POST request that returns **Status: 401 Unauthorized**. It is because we are still using an old password. So we are required to change the username and password with the new one. Provide the username and password which we have configured in the properties file. We get the Status: 201 Created.



Connecting RESTful Services to JPA

**Creating a User Entity and some test Data**

Let's create a User entity and a UserRepository so that we can access the detail of the user.

**Step 1:** Open **pom.xml** file and add **spring-boot-starter-data-jpa** dependency.

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-data-jpa</artifactId>
4. </dependency>

**Step 2:** Make the **User** class as an entity by adding an annotation **@Entity** just above the User class.

22M

439

History of Java

**@Entity:** Entities are nothing but POJO (Plain Old Java Object). It represents the data that can be persisted to the database. It represents a table in a database. Every instance of an entity represents a row in a table. We cannot declare an entity class as **final**.

**Step 3:** Make the Id primary key by adding an annotation @Id just above the Id variable. Also, add an annotation **@GeneratedValue**.

**@Id:** It defines that the member field below is the primary key of the current entity. Every entity must have a primary key that uniquely defines the column.

**@GeneratedValue:** The @GeneratedValue annotation may be applied to a primary key property or field of an entity with the **@Id** annotation. It is used to support the primary key. We have to add the **@GeneratedValue** annotation to the primary key attribute and choose a generation type. The default generation type is GenerationType.AUTO.

**User.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.Date;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.GeneratedValue;
5. **import** javax.persistence.Id;
6. **import** javax.validation.constraints.Past;
7. **import** javax.validation.constraints.Size;
8. **import** io.swagger.annotations.ApiModel;
9. **import** io.swagger.annotations.ApiModelProperty;
10. @ApiModel(description="All details about the user")
11. @Entity
12. **public** **class** User
13. {
14. //making Id as primary key
15. @Id
16. @GeneratedValue
17. **private** Integer id;
18. @Size(min=5, message="Name should have atleast 5 characters")
19. @ApiModelProperty(notes="name should have atleast 5 characters")
20. **private** String name;
21. @Past
22. @ApiModelProperty(notes="Birth date should be in the past")
23. **private** Date dob;
24. //default constructor
25. **protected** User()
26. {
28. }
29. **public** User(Integer id, String name, Date dob)
30. {
31. **super**();
32. **this**.id = id;
33. **this**.name = name;
34. **this**.dob = dob;
35. }
36. **public** Integer getId()
37. {
38. **return** id;
39. }
40. **public** **void** setId(Integer id)
41. {
42. **this**.id = id;
43. }
44. **public** String getName()
45. {
46. **return** name;
47. }
48. **public** **void** setName(String name)
49. {
50. **this**.name = name;
51. }
52. **public** Date getDob()
53. {
54. **return** dob;
55. }
56. **public** **void** setDob(Date dob)
57. {
58. **this**.dob = dob;
59. }
60. @Override
61. **public** String toString()
62. {
63. //return "User [id=" + id + ", name=" + name + ", dob=" + dob + "]";
64. **return** String.format("User [id=%s, name=%s, dob=%s]", id, name, dob);
65. }
66. }

Before moving to the next step, **remove** or **comment** the basic security dependency in the **pom.xml**.

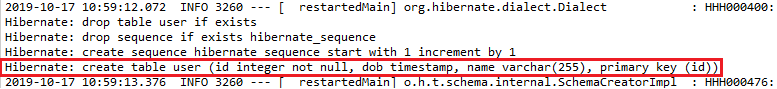
**Step 4:** Open **application.properties** file and enable the **H2 console** so that we can see what data we have inserted in the table.

1. spring.h2.console.enabled=**true**

We also need to enable the **SQL logging** to see which SQL statements are executing. It starts SQL logging in the log when the statement executes.

1. spring.jpa.show-sql=**true**

Now restart the application to pick up the changes. We can see in the following image that the table is created.



**Step 5:** We have to create a SQL file to insert data in the user table.

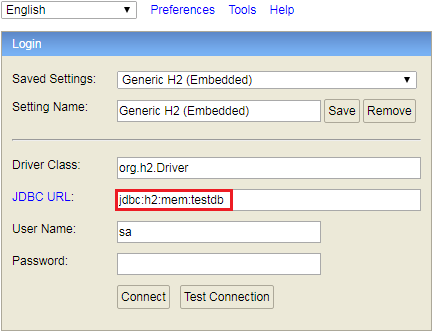
Right-click on the folder **src/main/resource** -> New ->Other -> Select SQL File -> provide the file name **data.sql ->** Click on finish.

**Step 6:** In the data.sql file, insert the data into **user** table. We have inserted the following data:

1. insert into user values(1, sysdate(), 'John');
2. insert into user values(2, sysdate(),'Robert');
3. insert into user values(3, sysdate(), 'Andrew');

Run the application.

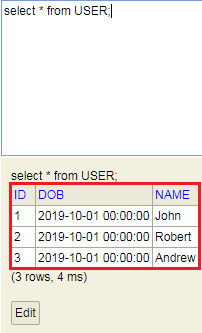
**Step 7:** Open the browser and type http://localhost:8080/h2-console to connect the H2 console. Make sure that it has JDBC URL: **jdbc:h2:mem:testdb**. Do not write anything in the password field.



**Step 8:** Click on the **Connect** button to login. The table which we have created appears on the left-hand side of the page.

**Step 10:** Type the query **select \* from user**;

It displays the table of records which we have inserted.



Updating GET Methods on User Resource to Use JPA

In this topic, we will create a service that retrieves all the users.

Still, we are using the **UserResource**, which talks to the in-memory. Now we will create a new UserResource that will talk to the embedded database. Let's create a new user resource.

**Step 1:** Copy the **UserResource.java** file and paste it in the user package. Rename it with UserJPAResource.

**Step 2:** Now, we have two URIs with the same name that create conflict. To remove this conflict, we will add **/jpa** in **UserJPAResource.java** file.

Difference between JDK, JRE, and JVM

**UserJPAResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** **static** org.springframework.hateoas.mvc.ControllerLinkBuilder.\*;
3. **import** java.net.URI;
4. **import** java.util.List;
5. **import** javax.validation.Valid;
6. **import** org.springframework.beans.factory.annotation.Autowired;
7. **import** org.springframework.hateoas.Resource;
8. **import** org.springframework.hateoas.mvc.ControllerLinkBuilder;
9. **import** org.springframework.http.ResponseEntity;
10. **import** org.springframework.web.bind.annotation.DeleteMapping;
11. **import** org.springframework.web.bind.annotation.GetMapping;
12. **import** org.springframework.web.bind.annotation.PathVariable;
13. **import** org.springframework.web.bind.annotation.PostMapping;
14. **import** org.springframework.web.bind.annotation.RequestBody;
15. **import** org.springframework.web.bind.annotation.RestController;
16. **import** org.springframework.web.servlet.support.ServletUriComponentsBuilder;
17. @RestController
18. **public** **class** UserJPAResource
19. {
20. @Autowired
21. **private** UserDaoService service;
22. @GetMapping("/jpa/users")
23. **public** List<User> retriveAllUsers()
24. {
25. **return** service.findAll();
26. }
27. @GetMapping("/jpa/users/{id}")
28. **public** Resource<User> retriveUser(@PathVariable **int** id)
29. {
30. User user= service.findOne(id);
31. **if**(user==**null**)
32. //runtime exception
33. **throw** **new** UserNotFoundException("id: "+ id);
34. //"all-users", SERVER\_PATH + "/users"
35. //retrieveAllUsers
36. Resource<User> resource=**new** Resource<User>(user);   //constructor of Resource class
37. //add link to retrieve all the users
38. ControllerLinkBuilder linkTo=linkTo(methodOn(**this**.getClass()).retriveAllUsers());
39. resource.add(linkTo.withRel("all-users"));
40. **return** resource;
41. }
42. //method that delete a user resource
43. @DeleteMapping("/jpa/users/{id}")
44. **public** **void** deleteUser(@PathVariable **int** id)
45. {
46. User user= service.deleteById(id);
47. **if**(user==**null**)
48. //runtime exception
49. **throw** **new** UserNotFoundException("id: "+ id);
50. }
51. //method that posts a new user detail and returns the status of the user resource
52. @PostMapping("/jpa/users")
53. **public** ResponseEntity<Object> createUser(@Valid @RequestBody User user)
54. {
55. User sevedUser=service.save(user);
56. URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(sevedUser.getId()).toUri();
57. **return** ResponseEntity.created(location).build();
58. }
59. }

But it is not really talking to the database. We need to create a spring data repository.

**Step 3:** Create an interface with the name **UserRepository** that extends JpaRepository. Specify the entity that has to be managed. We have specified User and Integer. Now we have the UserRepository ready.

**UserRepository.java**

1. **package** com.javatpoint.server.main.user;
2. **import** org.springframework.data.jpa.repository.JpaRepository;
3. **import** org.springframework.stereotype.Repository;
4. @Repository
5. **public** **interface** UserRepository **extends** JpaRepository<User, Integer>
6. {
8. }

**Step 4:** Create the use of UserResource. We have autowired the UserRepository interface in the UserJPAResource class.

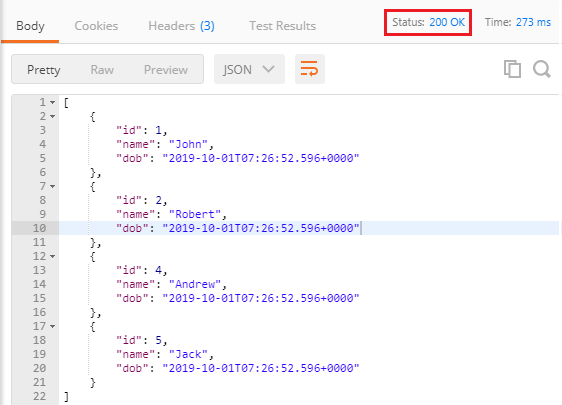
1. @Autowired
2. **private** UserRepository userRepository;

**Step 5:** Return the **userRepository.findAll()** in the retriveAllUsers() method.

1. @GetMapping("/jpa/users")
2. **public** List<User> retriveAllUsers()
3. {
4. **return** userRepository.findAll();
5. }

The retriveAllUsers() is the only method that retrieve data form the embedded database, all the other methods retrieve data from the static array list.

**Step 6:** Open the **Postman**. Type the URI http://localhost:8080/jpa/users and send a **GET** request. It shows all the data that is fetched from the embedded database.



Again send a GET request with the URL http://localhost:8080/jpa/users/1. It returns the specified user id, i.e. 1, but it picks up data from memory.

But we are required to fetch data from the embedded database. We need to change the following services in the **UserJPAResource.java**.

1. @GetMapping("/jpa/users")
2. **public** List<User> retriveAllUsers()
3. {
4. **return** userRepository.findAll();
5. }

In the following service, the **findById()** returns the Option of User whether user is null or not null. Whenever we use findById(), there is two **possibilities:** id exist or not exist. When it does not exist it comes with a proper object. We will check the user exist or not by the statement **if(!user.isPresent())**. If the user is not present it throws an exception.

1. @GetMapping("/jpa/users/{id}")
2. **public** Resource<User> retriveUser(@PathVariable **int** id)
3. {
4. Optional<User> user= userRepository.findById(id);
5. **if**(user.isPresent())
6. //runtime exception
7. **throw** **new** UserNotFoundException("id: "+ id);
8. //"all-users", SERVER\_PATH + "/users"
9. //retrieveAllUsers
10. Resource<User> resource=**new** Resource<User>(user.get()); //constructor of Resource class
11. //add link to retrieve all the users
12. ControllerLinkBuilder linkTo=linkTo(methodOn(**this**.getClass()).retriveAllUsers());
13. resource.add(linkTo.withRel("all-users"));
14. **return** resource;
15. }

Again send a **GET** request with the URL http://localhost:8080/jpa/users/1. It returns a specified user and the link to **/jpa/users**.

1. {
2. "name": "John",
3. "dob": "2019-10-01"T0726:52.596+0000",
4. "\_links": {
5. "all-users": {
6. "href": "http://localhost:8080/jpa/users"
7. }
8. }
9. }

Updating POST and DELETE methods on UserResource to use JPA

In this section, we will convert deleteUser() method and createUser() method to use JPA. Let's make the changes in the **UserJPAResource.java**.

**Step 1:** Change the service of the **deleteUser()** method.

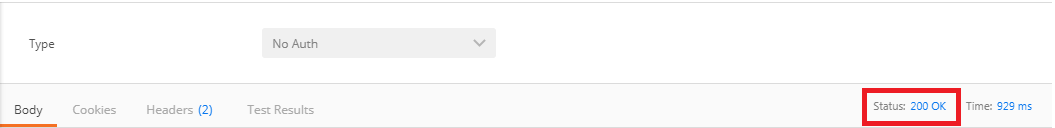
**Step 2:** Delete the return type because userRepository's delete() method does not return anything.

1. @DeleteMapping("/jpa/users/{id}")
2. **public** **void** deleteUser(@PathVariable **int** id)
3. {
4. userRepository.deleteById(id);
5. }

If it fails, it throws an exception.

OOPs Concepts in Java

**Step 3:** Open the **Postman** and send a **DELETE** request with the URL http://localhost:8080/jpa/users/1.



**Status: 200** OK denotes that record has been deleted successfully.

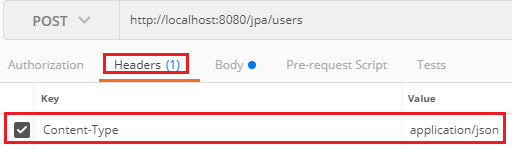
Again send a **DELETE** request with the same URL http://localhost:8080/jpa/users/1. It returns a message "**no entity with id 1 exist**".

1. {
2. "timesatmp": "2017-07-20T12:10:29.988+0000",
3. "message": "No class com.javatpoint.rest.webServices.restfulwebservices.user. User entity with id 1 exists! ",
4. "details": "uri=/jpa/user/1"
5. }

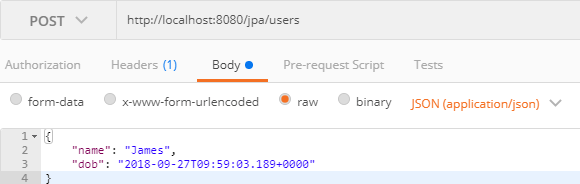
Now we will generate a **POST** request to create a user.

**Step 4:** Send a **POST** request with the URL http://localhost:8080/jpa/users.

**Step 5:** Click on the **Headers** tab and make sure that the **Content-Type** is **application/json**.



**Step 6:** Go to the Body tab and enter the **name** and **dob** of the user. We have entered the name **James**.



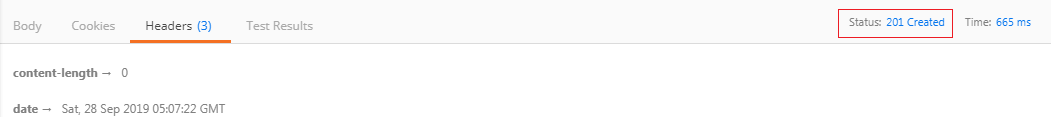
**Step 7:** Click on the Send button. When we try to create a user, it throws **ConstaintViolationException**.

It is because hibernate uses a sequence. In the User entity, Id is generated value, so it creates a sequence for us. Hibernate is trying to insert a row with id 1. It conflicts with the data that we already have. To remove this conflict, do the following:

**Step 8:** Open the **data.sql** file and change the Ids.

1. insert into user values(101, sysdate(), "John");
2. insert into user values(102, sysdate(), "Robert");
3. insert into user values(104, sysdate(), "Andrew");
4. insert into user values(105, sysdate(), "Jack");

**Step 9:** Again send a **POST** request. It returns the **Status: 201 Created**. The status shows that the user is created successfully.



Creating Post Entity and Many to One Relationship with User Entity

In this section, we will create a Post entity that contains many to one relationship with the User entity.

**Step 1:** Create a class with the name **Post.java** in the package **com.javatpoint.server.main.user**.

**Step 2:** Post.java is an entity, so we need to add **@Entity** annotation.

**Step 3:** Add three fields: **id, description,** and **user**.

32.8M

714

Exception Handling in Java - Javatpoint

1. **private** Integer id;
2. **private** String description;
3. **private** User user;

**Step 4:** Generate Getters and Setters.

**Step 5:** Generate toString().

**Remember:** Uncheck the **user** during the generation of toString().

**Step 6:** A user can do many posts, so add **@ManyToOne** annotation. The User entity has many to one relationship with the Post entity. The fetch type will not retrieve the details of the user unless we called **Post.getUser**.

1. @ManyToOne(fetch=FetchType.LAZY)

**Step 7:** Id is a primary key, so we need to add **@Id** annotation.

**Post.java**

1. **package** com.javatpoint.server.main.user;
2. **import** javax.persistence.Entity;
3. **import** javax.persistence.FetchType;
4. **import** javax.persistence.GeneratedValue;
5. **import** javax.persistence.Id;
6. **import** javax.persistence.ManyToOne;
7. @Entity
8. **public** **class** Post
9. {
10. @Id
11. @GeneratedValue
12. **private** Integer id;
13. **private** String description;
14. //many to one mapping
15. @ManyToOne(fetch=FetchType.LAZY)
16. **private** User user;
17. **public** Integer getId()
18. {
19. **return** id;
20. }
21. **public** **void** setId(Integer id)
22. {
23. **this**.id = id;
24. }
25. **public** String getDescription()
26. {
27. **return** description;
28. }
29. **public** **void** setDescription(String description)
30. {
31. description = description;
32. }
33. **public** User getUser()
34. {
35. **return** user;
36. }
37. **public** **void** setUser(User user)
38. {
39. **this**.user = user;
40. }
41. @Override
42. **public** String toString()
43. {
44. **return** String.format("Post [id=%s, description=%s]", id);
45. }
46. }

We have configured the relationship on the side of Post entity. Now we are required to configure relationship on the side of User entity.

A user can make a list of posts, so the post has one-to-many relationships.

**Step 8:** Open **User.java** file and create a list of posts.

1. **private** List<Post> posts

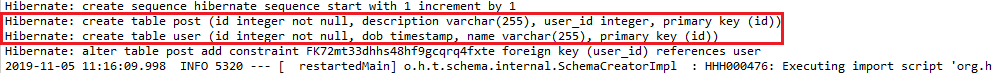
**Step 9:** Add an annotation **@OneToMany** with property **(mappedBy="user")**. It will create a relationship column in the Post entity.

**Step 10:** Generate Getters and Setters.

**User.java**

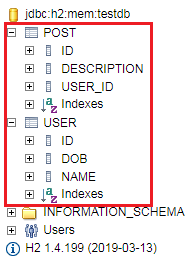
1. **package** com.javatpoint.server.main.user;
2. **import** java.util.Date;
3. **import** java.util.List;
4. **import** javax.persistence.Entity;
5. **import** javax.persistence.GeneratedValue;
6. **import** javax.persistence.Id;
7. **import** javax.persistence.OneToMany;
8. **import** javax.validation.constraints.Past;
9. **import** javax.validation.constraints.Size;
10. **import** io.swagger.annotations.ApiModel;
11. **import** io.swagger.annotations.ApiModelProperty;
12. @ApiModel(description="All details about the user")
13. @Entity
14. **public** **class** User
15. {
16. //Id as a primary key
17. @Id
18. @GeneratedValue
19. **private** Integer id;
20. @Size(min=5, message="Name should have atleast 5 characters")
21. @ApiModelProperty(notes="name should have atleast 5 characters")
22. **private** String name;
23. @Past
24. @ApiModelProperty(notes="Birth date should be in the past")
25. **private** Date dob;
26. //default constructor
27. @OneToMany(mappedBy="user")
28. **private** List<Post> posts;
29. **protected** User()
30. {
32. }
33. **public** User(Integer id, String name, Date dob)
34. {
35. **super**();
36. **this**.id = id;
37. **this**.name = name;
38. **this**.dob = dob;
39. }
40. **public** Integer getId()
41. {
42. **return** id;
43. }
44. **public** **void** setId(Integer id)
45. {
46. **this**.id = id;
47. }
48. **public** String getName()
49. {
50. **return** name;
51. }
52. **public** **void** setName(String name)
53. {
54. **this**.name = name;
55. }
56. **public** Date getDob()
57. {
58. **return** dob;
59. }
60. **public** **void** setDob(Date dob)
61. {
62. **this**.dob = dob;
63. }
64. **public** List<Post> getPosts()
65. {
66. **return** posts;
67. }
68. **public** **void** setPosts(List<Post> posts)
69. {
70. **this**.posts = posts;
71. }
72. @Override
73. **public** String toString()
74. {
75. //return "User [id=" + id + ", name=" + name + ", dob=" + dob + "]";
76. **return** String.format("User [id=%s, name=%s, dob=%s]", id, name, dob);
77. }
78. }

**Step 11:** Restart the application.



We can see in the log that there are two tables post and user. The Post table has a link to the User table by a user id. A user can have multiple posts, and all of them have the same user id.

**Step 12:** Now, open the **H2 Console**. We can see that are two tables named USER and POST.



**Step 13:** Open the **data.sql** file that we have created earlier and insert the data into **POST** table. We have inserted the following data:

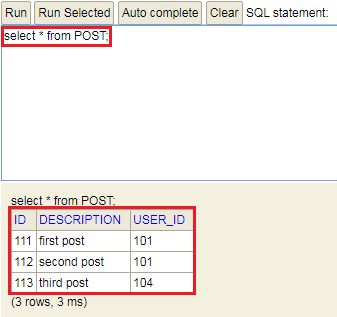
**data.sql**

1. insert into user values(101, sysdate(), 'John');
2. insert into user values(102, sysdate(), 'Robert');
3. insert into user values(104, sysdate(), 'Andrew');
4. insert into user values(105, sysdate(), 'Jack');
5. insert into post values(111, 'first post', 101);
6. insert into post values(112, 'second post', 101);
7. insert into post values(113, 'third post', 104);

**Step 14:** Restart the application.

**Step 15:** Restart the H2 Console and execute the query **SELECT \*FROM POST**;

It shows the data which we have inserted into the **data.sql** file.



Implementing a GET service to retrieve all Posts of a User

In this section, we will retrieve all the posts of a specific user.

**Step 1:** Open the **UserJPAResource.java** file and create a mapping for the URI "**/jpa/users/{id}/posts**"

1. @GetMapping("/jpa/users/{id}/posts")
2. **public** List<Post> retriveAllUsers(@PathVariable **int** id)
3. {
4. Optional<User> userOptional= userRepository.findById(id);
5. **if**(!userOptional.isPresent())
6. {
7. **throw** **new** UserNotFoundException("id: "+ id);
8. }
9. **return** userOptional.get().getPosts();
10. }

**Step 2:** There is no need to show user detail in the response, so we will add **@JsonIgnore** annotation just above the User field in **Post.java** file.

**Step 3:** Open the **Postman** and send a **GET** request with the URI http://localhost:8080/jpa/users/{id}/posts. In our case, we have specified user id **101**. It shows all the posts done by user 101.

Java Try Catch

1. [
2. {
3. "id": 111,
4. "description": "first post"
5. },
6. {
7. "id": 112,
8. "description": "second post"
9. }
10. ]

Now, we send a **GET** request for the user who has not created any post yet. The user 105 has not created any post so we will specify this user id in the URI http://localhost:8080/jpa/users/105/posts.

It shows a pair of **empty** square brackets. The brackets denote that the user exists, but the user has not created any post.

Again send a **GET** request for the user who does not exist in the database say **110**. It shows the **Status: 404 Not Found** with the following details:

1. {
2. "timesatmp": "2019-10-05T05:31:09.407+0000",
3. "message": "id-110",
4. "details": "uri=/jpa/users/110/posts"
5. }

Implementing POST Service to Create a Post for a User

In this section, we will enable post-operation to create a post for the specific user.

**Step 1:** Open the **UserJPAResource.java** file and create a PostMapping to create a post.

1. @PostMapping("/jpa/users/{id}/posts")
2. **public** ResponseEntity<Object> createUser(@PathVariable **int** id, @RequestBody Post post)
3. {
4. Optional<User> userOptional= userRepository.findById(id);
5. **if**(!userOptional.isPresent())
6. {
7. **throw** **new** UserNotFoundException("id: "+ id);
8. }
9. User user=userOptional.get();
10. //map the user to the post
11. post.setUser(user);
12. //save post to the database
13. postRepository.save(post);
14. //getting the path of the post and append id of the post to the URI
15. URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(post.getId()).toUri();
16. //returns the location of the created post
17. **return** ResponseEntity.created(location).build();
18. }

**Step 2:** Create a post repository.

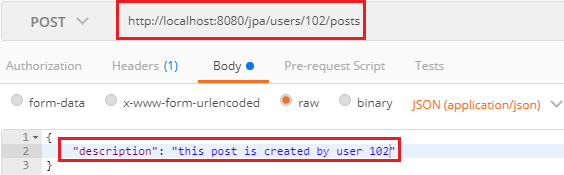
**PostRepository.java**

Competitive questions on Structures in Hindi

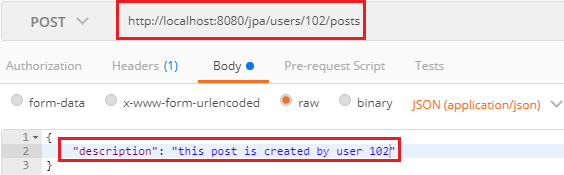
Keep Watching

1. **package** com.javatpoint.server.main.user;
2. **import** org.springframework.data.jpa.repository.JpaRepository;
3. **import** org.springframework.stereotype.Repository;
4. @Repository
5. **public** **interface** PostRepository **extends** JpaRepository<Post, Integer>
6. {
8. }

**Step 3:** Open the **Postman** and send a **POST** request with the URI http://localhost:8080/jpa/users/102/posts. Under the **Body** tab, insert the post description.



It returns the **Status: 201 Created**. We can also see this post in the database by executing the query **select \*from post;**



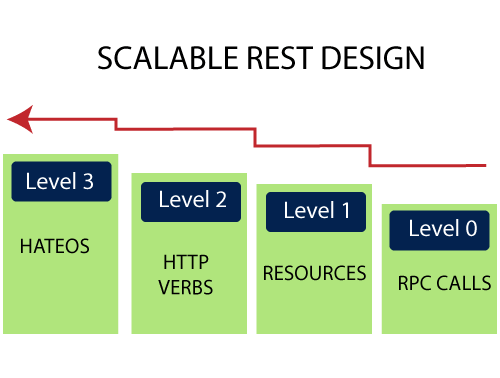
Richardson Maturity Model

**Richardson Maturity Model** grades API by their RESTful maturity. It is proposed by **Leonard Richardson**. The Richardson maturity model is a way to grade your API according to the constraints of REST. It breaks down the principal element of the REST approach into **four** levels (0 to 3).

There are four levels:

* Level 0: The Swamp of POX
* Level 1: Resources
* Level 2: HTTP Verbs
* Level 3: Hypermedia Control

For example, a level higher is more RESTful compared to one that is at a lower level. Only when an API reaches level 4, we consider it as a RESTful API.



Level 0: The Swamp of POX

Level 0 is also known as POX (Plain Old XML). At level 0, HTTP is used only as a transport protocol that is used as a remote interaction. It does not take the advantages of HTTP like different HTTP methods, and HTTP cache. To get and post the data, we send a request to the same URI, and only the POST method may be used. These APIs use only one URI and one HTTP method called POST. In short, it exposes SOAP web services in the REST style.

How to find Nth Highest Salary in SQL

**For example,** there can be many customers for a particular company. For all the different customers, we have only one endpoint. To do any of the operations like get, delete, update, we use the same POST method.

To get the data: http://localhost:8080/customer

To post the data: http://localhost:8080/customer

In the above two URIs, we have used the same URI and method to get and post the customers.

Level 1: Resources

When an API can distinguish between different resources, it might be at level 1. It uses multiple URIs. Where every URI is the entry point to a specific resource. It exposes resources with proper URI. Level 1 tackles complexity by breaking down huge service endpoints into multiple **different endpoints**. It also uses only one HTTP method POST for retrieving and creating data.

For example, if we want a list of specific products, we go through the URI http://localhost:8080/products. If we want a specific product, we go through the URI http://localhost:8080/products/mobile.

Remember the following points while building a URI:

* Use domain and subdomain to logically group or partition resources.
* Use **/** to indicate a hierarchical relationship.
* Use **,** and **;** to indicate non-hierarchical relationships.
* Use **-** and **\_** to improve the readability.
* Use **&** to separate parameters.
* Avoid including **file extensions**.

Level 2: HTTP Verbs

Level 2 indicates that an API must use the protocol properties to deal with scalability and failures. At level 2, correct **HTTP verbs** are used with each request. It suggests that in order to be truly RESTful, HTTP verbs must be used in API. For each of those requests, the correct HTTP response code is provided.

We don't use a single POST method for all requests. We use the **GET** method when we request a resource, and use the **DELETE** method when we want to delete a resource. Also, use the response codes of the application protocol.

For example, to get the customers, we send a request with the URI http://localhost:8080/customers, and the server sends proper response **200 OK**.

The following table shows the HTTP verbs and their usage:

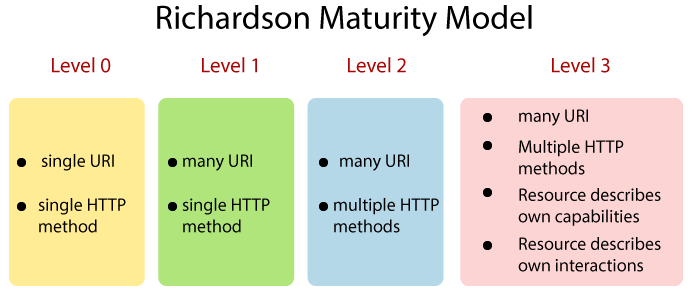
|  |  |  |
| --- | --- | --- |
| **Verbs** | **Safety & Idempotency** | **Usage** |
| GET | Y/Y | It retrieves the information. |
| POST | N/N | It is used to perform a variety of actions on the server, such as create a new resource, and update an existing resource, or making a mixture of changes to one or more resources. |
| DELETE | N/Y | It is used to delete a resource. |
| PUT | N/Y | It is used to update or replace an existing resource or to create a new resource with a URI specified by the client. |
| HEAD | Y/Y | It is used to retrieve the same headers as that of GET response but without any body in the response. |
| OPTIONS | Y/Y | It is used to find the list of HTTP methods supported by any resource or to ping the server. |
| TRACE | Y/Y | It is used for debugging, which echo's back headers that it has received. |

Level 3: Hypermedia Controls

Level 3 is the highest level. It is the combination of level 2 and HATEOAS. It also provides support for HATEOAS. It is helpful in self-documentation.

For example, if we send a GET request for customers, we will get a response for customers in JSON format with self-documenting Hypermedia.

The following figure shows the overview of the model:



# RESTful Web Services Best Practice

In this section, we will evaluate the best practice that must be followed while developing RESTful Web Services.

The first and last best practice is the **Consumer First**. It means, always think about your consumers. Before naming your resources, think from the perspective of the customers, what do they think about those resources? Will they able to understand these resources?

**Consumer First**

We must have excellent documentation for our API. Swagger is one of the most popular documentation standard for RESTful API. Make sure that our consumer understands the documentation that we have produced.

Java Try Catch

The next best practice is to make the best use of **HTTP**. RESTful web services are based on HTTP. Make the best use of the request methods. Use the right request method (**GET, POST, PUT, and DELETE**) appropriate for our specific action and ensure that we are sending a proper response status back.

For example, when a **ESOURCE NOT FOUND**, don't send the SERVER ERROR. When a resource is CREATED, don't send SUCCESS, send CREATED back.

Ensure that there is no secure information in the URI. Think about what you are putting in your URI. Ensure that there is nothing secure that is going in the URIs.

Always use plurals. In the previous examples, we have used /users instead of using /user. Similarly, for accessing a resource, we have used **/users/1 not /user/1**. It is more readable than using the singular.

When we think about resources, always use nouns for resources. But it is not always possible. There are always exception scenarios. For all these exceptions scenarios, define a consistent approach if we are searching through user use **/user/search**.

For example, if we put a star on the gists the Github sends the request to the resource of the **gists** (/gists/{id}) and sends a put request with the star in the URI.

# Spring Boot Interview Questions

### **1) What is Spring Boot?**

Spring Boot is a Spring module which provides RAD (Rapid Application Development) feature to Spring framework.

It is used to create stand alone spring based application that you can just run because it needs very little spring configuration.

For more information [click here.](https://www.javatpoint.com/spring-boot-introduction)

### **2) What are the advantages of Spring Boot?**

* Create stand-alone Spring applications that can be started using java -jar.
* Embed Tomcat, Jetty or Undertow directly. You don't need to deploy WAR files.
* It provides opinionated 'starter' POMs to simplify your Maven configuration.
* It automatically configure Spring whenever possible.

For more information [click here.](https://www.javatpoint.com/spring-boot-introduction)

32.6M

631

Difference between JDK, JRE, and JVM

### **3) What are the features of Spring Boot?**

* Web Development
* SpringApplication
* Application events and listeners
* Admin features

For more information [click here.](https://www.javatpoint.com/spring-boot-features)

### **4) How to create Spring Boot application using Maven?**

There are multiple approaches to create Spring Boot project. We can use any of the following approach to create application.

* Spring Maven Project
* Spring Starter Project Wizard
* Spring Initializr
* Spring Boot CLI

For more information [click here.](https://www.javatpoint.com/spring-maven-project)

### **5) How to create Spring Boot project using Spring Initializer?**

It is a web tool which is provided by Spring on official site. You can create Spring Boot project by providing project details.

For more information [click here.](https://www.javatpoint.com/spring-initializr)

### **6) How to create Spring Boot project using boot CLI?**

It is a tool which you can download from the official site of Spring Framework. Here, we are explaining steps.

Download the CLI tool from official site and For more information [click here.](https://www.javatpoint.com/spring-boot-cli)

### **7) How to create simple Spring Boot application?**

To create an application. We are using STS (Spring Tool Suite) IDE and it includes the various steps that are explaining in steps.

For more information [click here.](https://www.javatpoint.com/spring-boot-application)

### **8) What are the Spring Boot Annotations?**

The @RestController is a stereotype annotation. It adds @Controller and @ResponseBody annotations to the class. We need to import org.springframework.web.bind.annotation package in our file, in order to implement it.

For more information [click here.](https://www.javatpoint.com/spring-boot-annotations)

### **9) What is Spring Boot dependency management?**

Spring Boot manages dependencies and configuration automatically. You don't need to specify version for any of that dependencies.

Spring Boot upgrades all dependencies automatically when you upgrade Spring Boot.

For more information [click here.](https://www.javatpoint.com/spring-boot-dm)

### **10) What are the Spring Boot properties?**

Spring Boot provides various properties which can be specified inside our project's **application.properties** file. These properties have default values and you can set that inside the properties file. Properties are used to set values like: server-port number, database connection configuration etc.

For more information [click here.](https://www.javatpoint.com/spring-boot-properties)

### **11) What are the Spring Boot Starters?**

Starters are a set of convenient dependency descriptors which we can include in our application.

Spring Boot provides built-in starters which makes development easier and rapid. For example, if we want to get started using Spring and JPA for database access, just include the **spring-boot-starter-data-jpa** dependency in your project.

For more information [click here.](https://www.javatpoint.com/spring-boot-starters)

### **12) What is Spring Boot Actuator?**

Spring Boot provides actuator to monitor and manage our application. Actuator is a tool which has HTTP endpoints. when application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints.

For more information [click here.](https://www.javatpoint.com/spring-boot-actuator)

### **13) What is thymeleaf?**

It is a server side Java template engine for web application. It's main goal is to bring elegant natural templates to your web application.

It can be integrate with Spring Framework and ideal for HTML5 Java web applications.

For more information [click here.](https://www.javatpoint.com/spring-boot-thymeleaf-view)

### **14) How to use thymeleaf?**

In order to use Thymeleaf we must add it into our pom.xml file like:

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-thymeleaf**</artifactId>**
4. **</dependency>**

For more information [click here.](https://www.javatpoint.com/spring-boot-thymeleaf-view)

### **15) How to connect Spring Boot to the database using JPA?**

Spring Boot provides **spring-boot-starter-data-jpa** starter to connect Spring application with relational database efficiently. You can use it into project POM (Project Object Model) file.

For more information [click here.](https://www.javatpoint.com/spring-boot-jpa)

### **16) How to connect Spring Boot application to database using JDBC?**

Spring Boot provides starter and libraries for connecting to our application with JDBC. Here, we are creating an application which connects with Mysql database. It includes the following steps to create and setup JDBC with Spring Boot.

For more information [click here.](https://www.javatpoint.com/spring-boot-jdbc)

### **17) What is @RestController annotation in Spring Boot?**

The @RestController is a stereotype annotation. It adds @Controller and @ResponseBody annotations to the class. We need to import org.springframework.web.bind.annotation package in our file, in order to implement it.

For more information [click here.](https://www.javatpoint.com/spring-boot-annotations)

### **18) What is @RequestMapping annotation in Spring Boot?**

The **@RequestMapping** annotation is used to provide routing information. It tells to the Spring that any HTTP request should map to the corresponding method. We need to import org.springframework.web.annotation package in our file.

For more information [click here.](https://www.javatpoint.com/spring-boot-annotations)

### **19) How to create Spring Boot application using Spring Starter Project Wizard?**

There is one more way to create Spring Boot project in STS (Spring Tool Suite). Creating project by using IDE is always a convenient way. Follow the following steps in order to create a Spring Boot Application by using this wizard.

For more information [click here.](https://www.javatpoint.com/spring-starter-project-wizard)

### **20) Spring Vs Spring Boot?**

Spring is a web application framework based on Java. It provides tools and libraries to create a complete cutomized web application.

Wheras Spring Boot is a spring module which is used to create spring application project that can just run.