



INSTITUTO TÉCNICO  
DE ESTUDIOS  
PROFESIONALES

Técnico superior en desarrollo de  
aplicaciones multiplataforma

# ACCESO A DATOS

---

Ciclo formativo segundo curso

A decorative element consisting of several horizontal and vertical red lines forming a grid pattern, centered behind the text.

CARLOS BLANCO GÓMEZ



## Contenido

<b>Herramientas de mapeo objeto-relacional (ORM)</b> .....	7
<b>Spring Boot</b> .....	7
<b>¿Qué es Micro Service?</b> .....	7
Ventajas.....	7
<b>¿Qué es Spring Boot?</b> .....	7
Ventajas.....	8
Metas.....	8
<b>¿Por qué Spring Boot?</b> .....	8
<b>¿Como funciona?</b> .....	8
<b>Arrancadores Spring Boot</b> .....	9
<b>Ejemplos</b> .....	9
<b>Configuración automática</b> .....	10
<b>Aplicación Spring Boot</b> .....	10
<b>Escaneo de componentes</b> .....	11
<b>Arquitectura Spring Boot</b> .....	11
<b>Arquitectura Spring Boot Flow</b> .....	13
<b>Creando Proyecto</b> .....	13
<b>Spring Initializr</b> .....	13
<b>Módulos Spring Initializr</b> .....	14
<b>Interfaz compatible</b> .....	15
<b>Generando un proyecto</b> .....	16
<b>Descargue e instale STS IDE</b> .....	17
<b>IDE de Spring Tool Suite (STS)</b> .....	17
<b>Instalación de STS</b> .....	18
<b>Crear un proyecto de Spring Boot</b> .....	20
<b>Spring Boot CLI</b> .....	25
<b>Spring Boot CLI</b> .....	25
<b>Creación de un proyecto de Spring Boot usando STS</b> .....	29
<b>Anotaciones de Spring Boot</b> .....	37
<b>Anotaciones de Core Spring Framework</b> .....	37
<b>Anotaciones de estereotipo de Spring Framework</b> .....	41
<b>Anotaciones de Spring Boot</b> .....	43





<b>Spring MVC y anotaciones REST.....</b>	43
<b>Gestión de dependencias de Spring Boot.....</b>	46
<b>Ventajas de la gestión de dependencias .....</b>	46
<b>Sistema de gestión de dependencias de Maven.....</b>	46
<b>Heredar padre inicial .....</b>	47
<b>Cambiar la versión de Java .....</b>	47
Adición del complemento Spring Boot Maven .....	47
<b>Spring Boot sin Parent POM .....</b>	48
<b>Propiedades de la aplicación Spring Boot .....</b>	49
<b>Ejemplo de application.properties.....</b>	51
<b>Categorías de propiedades de Spring Boot.....</b>	52
<b>Tabla de propiedades de la aplicación.....</b>	53
<b>Arrancadores Spring Boot.....</b>	58
<b>Entrantes de terceros.....</b>	59
Iniciadores de producción Spring Boot .....	61
Arrancadores técnicos Spring Boot.....	61
<b>Parent de Spring Boot Starter.....</b>	62
<b>Parent de Spring Boot Starter.....</b>	62
<b>Spring Boot Starter para parent interno.....</b>	63
<b>Spring Boot Starter sin parent.....</b>	65
<b>Spring Boot Starter Web .....</b>	66
<b>Servidor web incorporado Spring Boot .....</b>	67
<b>Usando otro servidor web incorporado .....</b>	68
<b>Servidor Jetty .....</b>	68
<b>Servidor Undertow .....</b>	69
<b>spring-boot-starter-web frente a spring-boot-starter-tomcat.....</b>	70
<b>Spring Data JPA .....</b>	72
<b>Spring Data JPA .....</b>	72
Características de Spring Data JPA.....	72
<b>Repositorio de datos de Spring.....</b>	74
<b>Spring Boot Starter Data JPA .....</b>	74
<b>Hibernar frente a JPA .....</b>	75
<b>Ejemplo de Spring Boot JPA.....</b>	76





<b>Starter Actuator .....</b>	81
<b>Características del actuador de arranque .....</b>	82
<b>Habilitación del actuador de arranque .....</b>	82
<b>Puntos finales del actuador de arranque.....</b>	83
<b>Propiedades del actuador Spring Boot .....</b>	84
<b>Ejemplo de actuador de arranque de resorte .....</b>	85
<b>Prueba de arranque de Spring Boot.....</b>	88
<b>Ejemplo de prueba de Spring Boot Starter .....</b>	89
<b>Spring Boot DevTools.....</b>	92
<b>Spring Boot DevTools.....</b>	92
<b>Funciones de Spring Boot DevTools .....</b>	92
<b>Recordar:.....</b>	93
<b>Recordar .....</b>	95
<b>Usando un archivo de activación .....</b>	95
<b>Ejemplo de Spring Boot DevTools .....</b>	95
<b>Proyecto Spring Boot Multi-Module .....</b>	97
<b>Proyecto de varios módulos .....</b>	97
<b>POM padre.....</b>	98
<b>Por qué necesitamos un proyecto de varios módulos .....</b>	98
<b>Módulo hijo module-ear, war, y jar .....</b>	98
<b>Proyectos / módulos secundarios de Maven.....</b>	99
<b>Estructura de directorio de proyectos de varios módulos .....</b>	99
<b>Ejemplo de proyecto de módulo múltiple Spring Boot .....</b>	105
<b>Agregador de POM (POM principal) .....</b>	106
<b>Spring Boot Packaging .....</b>	127
<b>WAR .....</b>	128
<b>JAR .....</b>	129
<b>EAR .....</b>	129
<b>Spring Boot Auto-configuration .....</b>	129
<b>Necesidad de autoconfiguración .....</b>	131
<b>Deshabilitar clases de auto-configuration.....</b>	132
<b>Ejemplo de configuración automática de Spring Boot .....</b>	133
<b>Depuración de la configuración automática .....</b>	140





<b>Ejemplo de Spring Boot Hello World .....</b>	146
<b>Implementación de proyectos con Tomcat.....</b>	150
<b>Ejemplo .....</b>	150
<b>Crear un WAR de Spring Boot .....</b>	153
<b>Implementar el archivo WAR en Tomcat.....</b>	159
<b>Spring Boot AOP .....</b>	161
<b>AOP .....</b>	161
<b>Beneficios de AOP .....</b>	162
<b>Preocupación transversal .....</b>	162
<b>Terminología AOP .....</b>	162
<b>AOP frente a OOP .....</b>	163
<b>Spring AOP frente a AspectJ .....</b>	164
<b>Tipos de advices de AOP .....</b>	165
<b>Spring Boot Starter AOP .....</b>	166
<b>Spring Boot AOP Before Advice .....</b>	166
<b>Ejemplo de Spring Boot antes de asesoramiento .....</b>	166
<b>Spring Boot AOP After Advice.....</b>	176
<b>Spring Boot after advice en un ejemplo .....</b>	176
<b>Spring Boot AOP Around Advice.....</b>	186
<b>Ejemplo de consejo de Spring Boot Around .....</b>	186
<b>Spring Boot AOP After Returning Advice.....</b>	194
<b>Spring Boot ejemplo de After Returning Advice .....</b>	194
<b>Spring Boot AOP After Throwing Advice.....</b>	203
<b>Spring Boot ejemplo de After Throwing Advice .....</b>	204
<b>Spring Boot JPA .....</b>	213
<b>¿Qué es JPA?.....</b>	213
<b>¿Por qué deberíamos usar JPA? .....</b>	214
<b>Funciones de JPA .....</b>	214
<b>Arquitectura JPA .....</b>	215
<b>Relaciones de clase JPA .....</b>	216
<b>Implementaciones JPA .....</b>	217
<b>Mapeo de relación de objeto (ORM) .....</b>	218
<b>Versiones JPA.....</b>	218





<b>Diferencia entre JPA e Hibernate .....</b>	219
<b>Spring Boot Starter Data JPA .....</b>	220
<b>Ejemplo de Spring Boot JPA.....</b>	220
<b>Spring Boot JDBC .....</b>	229
<b>Agrupación de conexiones JDBC.....</b>	230
<b>HikariCP .....</b>	231
<b>¿Por qué deberíamos usar Spring Boot JDBC? .....</b>	234
<b>JDBC frente a hibernación.....</b>	235
<b>Ejemplo de Spring Boot JDBC.....</b>	235
<b>Ejecuta la aplicación .....</b>	239
<b>Base de datos Spring Boot H2.....</b>	240
<b>¿Qué es la base de datos en memoria? .....</b>	240
<b>Persistencia frente a base de datos en memoria.....</b>	241
<b>¿Qué es la base de datos H2? .....</b>	241
<b>Configurar la base de datos H2 .....</b>	241
<b>Conservar los datos en la base de datos H2 .....</b>	242
<b>Crear esquema y completar datos.....</b>	243
<b>Consola H2.....</b>	243
<b>Ejemplo de Spring Boot H2 .....</b>	244
<b>Operaciones Spring Boot CRUD .....</b>	259
<b>¿Qué es la operación CRUD? .....</b>	259
<b>Operación CRUD estándar.....</b>	260
<b>Cómo funcionan las operaciones CRUD .....</b>	260
<b>Spring Boot CrudRepository.....</b>	261
<b>Spring Boot JpaRepository .....</b>	262
<b>¿Por qué deberíamos utilizar estas interfaces? .....</b>	263
<b>CrudRepository frente a JpaRepository .....</b>	263
<b>Ejemplo de operación Spring Boot CRUD .....</b>	264
<b>Spring Boot Thymeleaf .....</b>	279
<b>¿Qué es Thymeleaf?.....</b>	279
<b>¿Por qué usamos Thymeleaf? .....</b>	280
<b>¿Qué tipo de plantillas puede procesar Thymeleaf? .....</b>	280
<b>El dialecto estándar .....</b>	281





Características de Thymeleaf.....	283
Implementación de Thymeleaf.....	283
Ejemplo de Spring Boot Thymeleaf.....	283



## **Herramientas de mapeo objeto-relacional (ORM)**

### **Spring Boot**

Spring Boot es un marco de código abierto basado en Java que se utiliza para crear un microservicio. Está desarrollado por Pivotal Team y se utiliza para construir aplicaciones de resortes independientes y listas para producción. Este capítulo le dará una introducción a Spring Boot y lo familiarizará con sus conceptos básicos.

#### **¿Qué es Micro Service?**

Micro Service es una arquitectura que permite a los desarrolladores desarrollar e implementar servicios de forma independiente. Cada servicio en ejecución tiene su propio proceso y esto logra el modelo liviano para admitir aplicaciones comerciales.

#### Ventajas

Los microservicios ofrecen las siguientes ventajas a sus desarrolladores:

- Despliegue sencillo
- Escalabilidad simple
- Compatible con contenedores
- Configuración mínima
- Menor tiempo de producción

#### **¿Qué es Spring Boot?**

Spring Boot proporciona una buena plataforma para que los desarrolladores de Java desarrollen una aplicación de primavera independiente y de grado de



producción que puede **ejecutar**. Puede comenzar con configuraciones mínimas sin la necesidad de una configuración completa de Spring.

#### Ventajas

Spring Boot ofrece las siguientes ventajas a sus desarrolladores:

- Aplicaciones de muelles fáciles de entender y desarrollar
- Aumenta la productividad
- Reduce el tiempo de desarrollo

#### Metas

Spring Boot está diseñado con los siguientes objetivos:

- Para evitar una configuración XML compleja en Spring
- Desarrollar aplicaciones Spring listas para producción de una manera más sencilla
- Para reducir el tiempo de desarrollo y ejecutar la aplicación de forma independiente
- Ofrecer una forma más sencilla de comenzar con la aplicación.

## ¿Por qué Spring Boot?

Puede elegir Spring Boot debido a las características y beneficios que ofrece como se indica aquí:

- Proporciona una forma flexible de configurar Java Beans, configuraciones XML y transacciones de bases de datos.
- Proporciona un potente procesamiento por lotes y gestiona los puntos finales REST.
- En Spring Boot, todo se configura automáticamente; no se necesitan configuraciones manuales.
- Ofrece una aplicación de resorte basada en anotaciones.
- Facilita la gestión de la dependencia
- Incluye contenedor de servlet integrado

## ¿Como funciona?

Spring Boot configura automáticamente su aplicación en función de las dependencias que ha agregado al proyecto mediante la anotación **@EnableAutoConfiguration**. Por ejemplo, si la base de datos MySQL está en su classpath, pero no ha configurado ninguna conexión de base de datos, Spring Boot configura automáticamente una base de datos en memoria.





El punto de entrada de la aplicación Spring Boot es la clase que contiene la anotación **@SpringBootApplication** y el método principal.

Spring Boot escanea automáticamente todos los componentes incluidos en el proyecto usando la anotación **@ComponentScan**.

## Arrancadores Spring Boot

Manejar la gestión de la dependencia es una tarea difícil para los grandes proyectos. Spring Boot resuelve este problema proporcionando un conjunto de dependencias para la conveniencia de los desarrolladores.

Por ejemplo, si desea utilizar Spring y JPA para el acceso a la base de datos, es suficiente si incluye la dependencia **spring-boot-starter-data-jpa** en su proyecto.

Tenga en cuenta que todos los arrancadores Spring Boot siguen el mismo patrón de nomenclatura **spring-boot-starter-\***, donde \* indica que es un tipo de aplicación.

### Ejemplos

Mire los siguientes arrancadores de Spring Boot que se explican a continuación para una mejor comprensión:

**La dependencia de Spring Boot Starter Actuator** se usa para monitorear y administrar su aplicación. Su código se muestra a continuación:

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

**La dependencia de Spring Boot Starter Security** se usa para Spring Security. Su código se muestra a continuación:

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

**La dependencia web de Spring Boot Starter** se usa para escribir un Rest Endpoints. Su código se muestra a continuación:

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

**La dependencia de Spring Boot Starter Thyme Leaf** se utiliza para crear una aplicación web. Su código se muestra a continuación:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```



```
<artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

**La dependencia de Spring Boot Starter Test** se usa para escribir casos de prueba. Su código se muestra a continuación:

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

## Configuración automática

Spring Boot Auto Configuration configura automáticamente su aplicación Spring en función de las dependencias JAR que agregó en el proyecto. Por ejemplo, si la base de datos MySQL está en su ruta de clases, pero no ha configurado ninguna conexión de base de datos, Spring Boot configura automáticamente una base de datos en memoria.

Para este propósito, es necesario agregar **@EnableAutoConfiguration** anotación o **@SpringBootApplication** anotación a su archivo de clase principal. Luego, su aplicación Spring Boot se configurará automáticamente.

Observe el siguiente código para una mejor comprensión:

```
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.EnableAutoConfiguration;

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

## Aplicación Spring Boot

El punto de entrada de la aplicación Spring Boot es la clase que contiene la anotación **@SpringBootApplication**. Esta clase debe tener el método principal para ejecutar la aplicación Spring Boot. La anotación de **@SpringBootApplication** incluye la configuración automática, el escaneo de componentes y la configuración de Spring Boot.

Si agregó la anotación **@SpringBootApplication** a la clase, no necesita agregar la anotación **@EnableAutoConfiguration**, **@ComponentScan** y **@SpringBootConfiguration**. La anotación **@SpringBootApplication** incluye todas las demás anotaciones.

Observe el siguiente código para una mejor comprensión:





```
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

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

## Escaneo de componentes

La aplicación Spring Boot analiza todos los beans y las declaraciones de paquetes cuando la aplicación se inicializa. **Debe** agregar la anotación **@ComponentScan** para su archivo de clase para escanear sus componentes agregados en su proyecto.

Observe el siguiente código para una mejor comprensión:

```
import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;

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

## Arquitectura Spring Boot

Spring Boot es un módulo de Spring Framework. Se utiliza para crear aplicaciones independientes basadas en resortes de grado de producción con un esfuerzo mínimo. Está desarrollado sobre el núcleo Spring Framework.

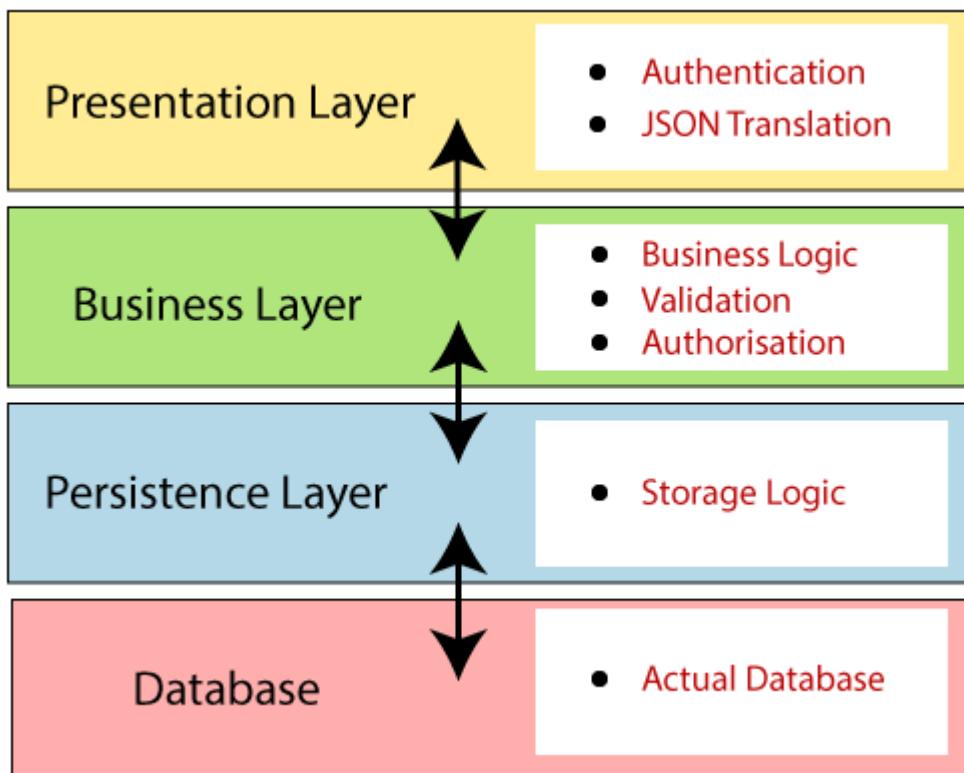
Spring Boot sigue una arquitectura en capas en la que cada capa se comunica con la capa directamente debajo o encima (estructura jerárquica).

Antes de comprender la **arquitectura Spring Boot**, debemos conocer las diferentes capas y clases presentes en ella. Hay **cuatro** capas en Spring Boot que son las siguientes:

- **Capa de presentación**
- **Capa empresarial**



- **Capa de persistencia**
  - **Capa de base de datos**



**Capa de presentación:** la capa de presentación maneja las solicitudes HTTP, traduce el parámetro JSON en un objeto, autentica la solicitud y la transfiere a la capa empresarial. En resumen, consta de **vistas**, es decir, parte frontal.

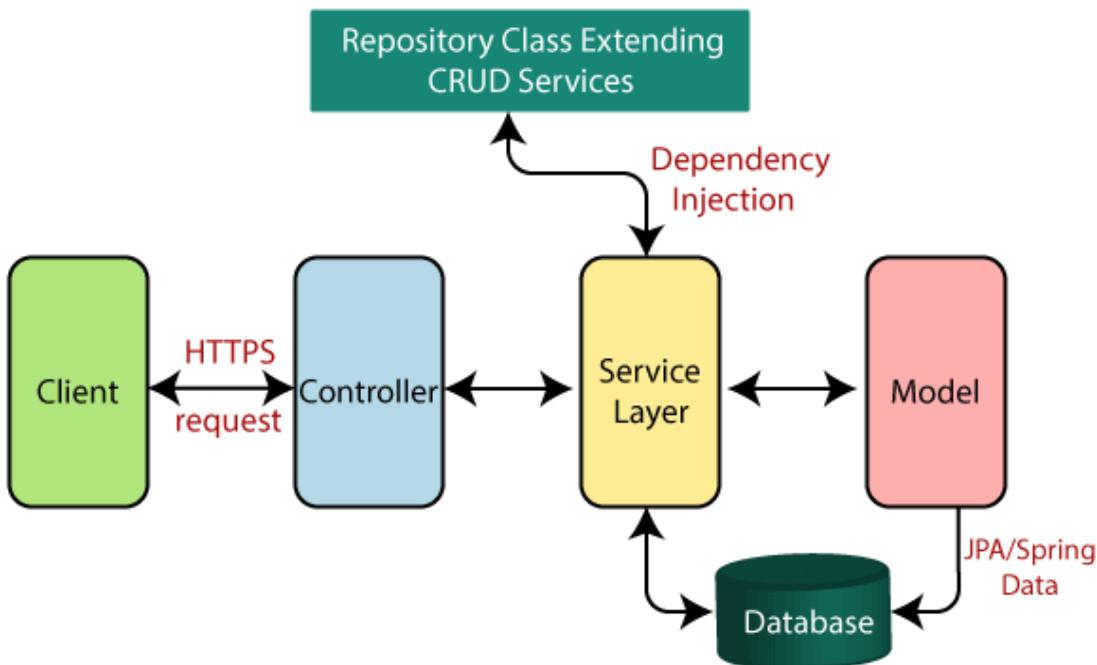
**Capa empresarial:** la capa empresarial maneja toda la **lógica empresarial**. Consiste en clases de servicio y utiliza servicios proporcionados por capas de acceso a datos. También realiza **autorización** y **validación**.

**Capa de persistencia:** la capa de persistencia contiene toda la **lógica de almacenamiento** y traduce los objetos comerciales desde y hacia las filas de la base de datos.

**Capa de base de datos:** En la capa de base de datos, se realizan operaciones **CRUD** (crear, recuperar, actualizar, eliminar).

## Arquitectura Spring Boot Flow

Spring Boot flow architecture



- Ahora tenemos clases de validación, clases de vista y clases de utilidad.
- Spring Boot usa todos los módulos de Spring MVC, Spring Data, etc. La arquitectura de Spring Boot es la misma que la arquitectura de Spring MVC, excepto una cosa: no hay necesidad de clases **DAO** y **DAOImpl** en Spring boot.
- Crea una capa de acceso a datos y realiza la operación CRUD.
- El cliente realiza las solicitudes HTTP (PUT o GET).
- La solicitud va al controlador, y el controlador asigna esa solicitud y la maneja. Después de eso, llama a la lógica de servicio si es necesario.
- En la capa de servicio, funciona toda la lógica empresarial. Realiza la lógica en los datos que se asignan a JPA con clases de modelo.
- Se devuelve una página JSP al usuario si no se produce ningún error.

## Creando Proyecto Spring Initializr

**Spring Initializr** es una **herramienta basada en web** proporcionada por Pivotal Web Service. Con la ayuda de **Spring Initializr**, podemos generar fácilmente la





estructura del **Spring Boot Project**. Ofrece API extensible para crear proyectos basados en JVM.

También proporciona varias opciones para el proyecto que se expresan en un modelo de metadatos. El modelo de metadatos nos permite configurar la lista de dependencias soportadas por JVM y versiones de plataforma, etc. Sirve sus metadatos de una manera notoria que brinda la asistencia necesaria a clientes de terceros.

## Módulos Spring Initializr

Spring Initializr tiene el siguiente módulo:

- **initializr-actuator:** Proporciona información adicional y estadísticas sobre la generación de proyectos. Es un módulo opcional.
- **initializr-bom:** En este módulo, **BOM** significa **Lista de materiales**. En Spring Boot, BOM es un tipo especial de **POM** que se utiliza para controlar las **versiones** de las **dependencias** de un proyecto. Proporciona un lugar central para definir y actualizar esas versiones. Proporciona flexibilidad para agregar una dependencia en nuestro módulo sin preocuparse por las versiones. Fuera del mundo del software, la **lista de materiales** es una lista de piezas, elementos, ensamblajes y otros materiales necesarios para crear productos. Explica **qué, cómo y dónde** recolectar los materiales requeridos.
- **initializr-docs:** proporciona documentación.
- **initializr-generator:** Es una biblioteca central de generación de proyectos.
- **initializr-generator-spring:**
- **initializr-generator-test:** proporciona una infraestructura de prueba para la generación de proyectos.
- **initializr-metadata:** proporciona una infraestructura de metadatos para varios aspectos de los proyectos.
- **initializr-service-example:** proporciona instancias personalizadas.
- **initializr-version-resolver:** es un módulo opcional para extraer números de versión de un POM arbitrario.
- **initializr-web:** proporciona puntos finales web para clientes de terceros.



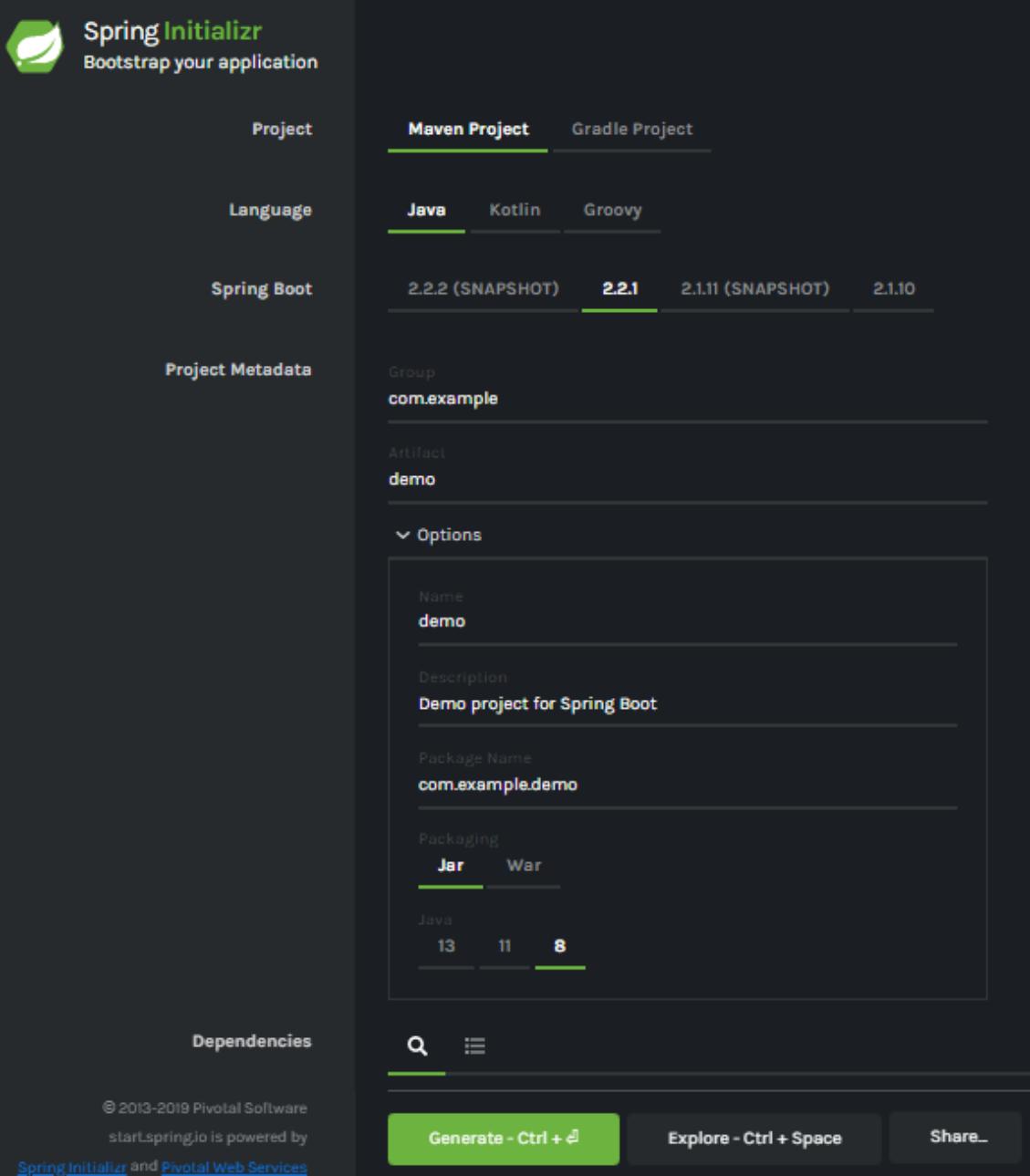


## Interfaz compatible

- Es compatible con **IDE STS, IntelliJ IDEA Ultimate, NetBeans, Eclipse**. Puede descargar el complemento desde <https://github.com/AlexFalappa/nb-springboot>. Si está utilizando VSCode, descargue el complemento desde <https://github.com/microsoft/vscode-spring-initializr>.
- Utilice la interfaz de usuario web personalizada <http://start.spring.io> o <https://start-scs.cfapps.io>.
- También es compatible con la línea de comandos con **Spring Boot CLI** o **cURL** o **HTTPie**.

La siguiente imagen muestra la interfaz de usuario de Spring Initializr:





The screenshot shows the Spring Initializr interface for creating a Maven project. The configuration includes:

- Project Type:** Maven Project
- Language:** Java
- Spring Boot Version:** 2.2.1
- Project Metadata:**
  - Group: com.example
  - Artifact: demo
  - Options:**
    - Name: demo
    - Description: Demo project for Spring Boot
    - Package Name: com.example.demo
    - Packaging: Jar
    - Java Version: 8
- Dependencies:** None listed.
- Bottom Buttons:** Generate - Ctrl + ⌘, Explore - Ctrl + Space, Share...

## Generando un proyecto

Antes de crear un proyecto, debemos ser amigables con la interfaz de usuario. La interfaz de usuario de Spring Initializr tiene las siguientes etiquetas:

- **Proyecto:** Define el **tipo** de proyecto. Podemos crear un **Proyecto Maven** o un **Proyecto Gradle**. Crearemos un **Proyecto Maven a lo** largo del manual.



- **Idioma:** Spring Initializr ofrece la posibilidad de elegir entre tres idiomas **Java**, **Kotlin** y **Groovy**. Java está seleccionado de forma predeterminada.
- **Spring Boot:** Podemos seleccionar la **versión** Spring Boot. La última versión es la **2.2.2**.
- **Metadatos del proyecto:** contiene información relacionada con el proyecto, como **Grupo**, Artefacto, etc. Grupo indica el nombre del **paquete**; **Artifact** denota el nombre de la **aplicación**. El nombre de grupo predeterminado es **com.example** y el nombre de artefacto predeterminado es **demo**.
- **Dependencias:** las dependencias son la colección de artefactos que podemos agregar a nuestro proyecto.

Hay otra sección de **Opciones** que contiene los siguientes campos:

- **Nombre:** es lo mismo que **Artifact**.
- **Descripción:** En el campo de descripción, podemos escribir una **descripción** del proyecto.
- **Nombre del paquete:** también es similar al nombre del **grupo**.
- **Packaging:** Podemos seleccionar el **paquete** del proyecto. Podemos elegir **Jar** o **War**.
- **Java:** Podemos seleccionar la versión de **JVM** que queremos usar. Usaremos la versión **de Java 8 a lo largo** del tutorial.

Hay un botón **Generar**. Cuando hacemos clic en el botón, comienza a empaquetar el proyecto y descarga el archivo **Jar** o **War**, que ha seleccionado.

## Descargue e instale STS IDE IDE de Spring Tool Suite (STS)

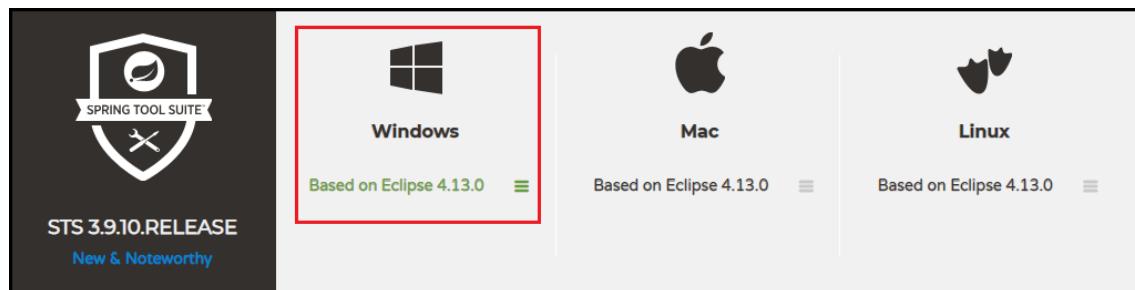
Spring Tool Suite es un IDE para desarrollar aplicaciones Spring. Es un entorno de desarrollo basado en Eclipse. Proporciona un entorno listo para usar para implementar, ejecutar, implementar y depurar la aplicación. Valida nuestra aplicación y proporciona soluciones rápidas para las aplicaciones.





## Instalación de STS

**Paso 1:** Descargue Spring Tool Suite desde <https://spring.io/tools3/sts/all> . Haga clic en la plataforma que está utilizando. En este tutorial, usamos la plataforma Windows.



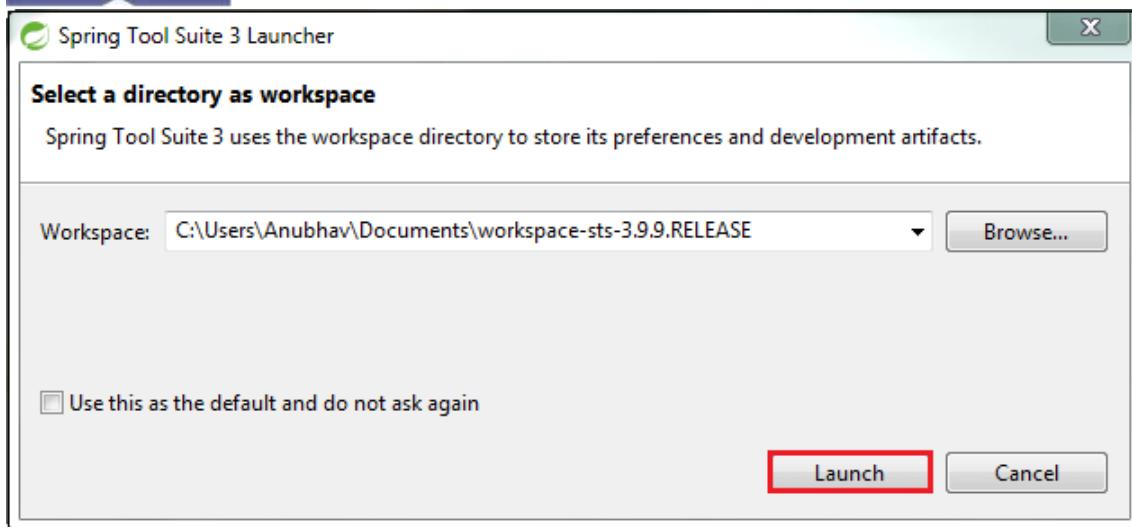
**Paso 2:** extraiga el archivo **zip** e instale el STS.

sts-bundle -> sts-3.9.9.RELEASE -> Haga doble clic en **STS.exe** .

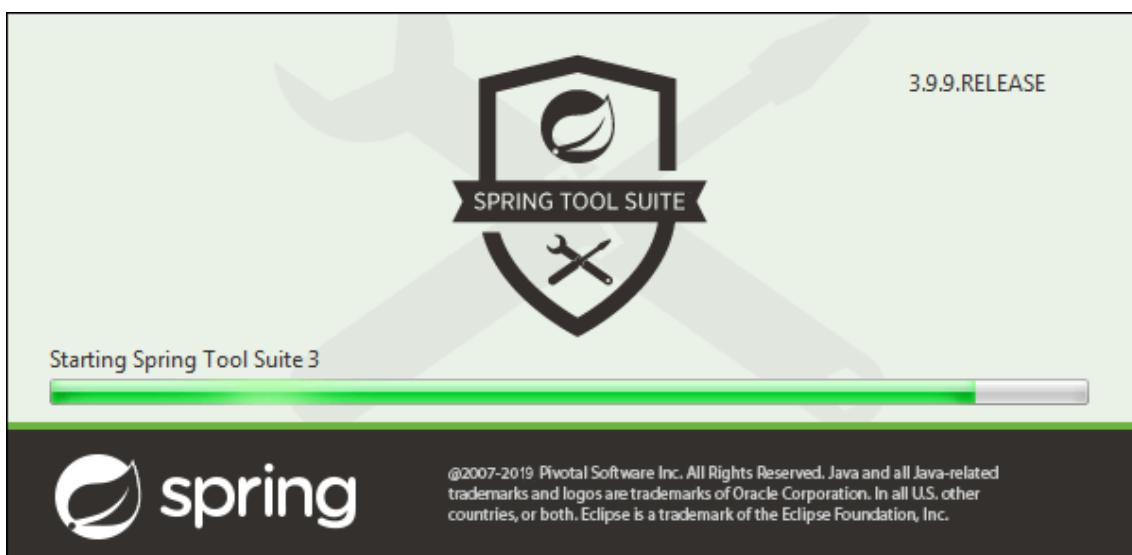


**Paso 3:** Aparece el cuadro de diálogo Spring Tool Suite 3 Launcher en la pantalla. Haga clic en el botón **Iniciar** . Puede cambiar el espacio de trabajo si lo desea.



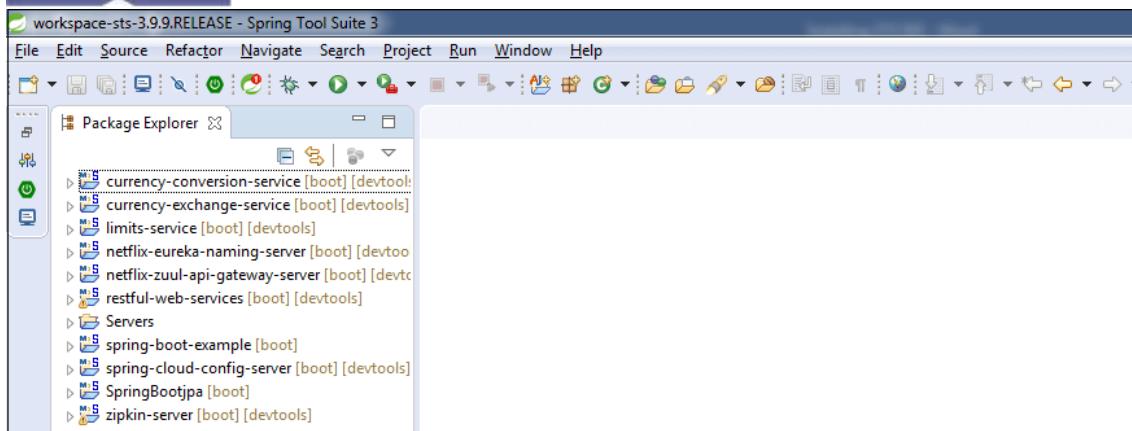


**Paso 4:** Comienza a lanzar el STS.



La interfaz de usuario de STS tiene el siguiente aspecto:





## Crear un proyecto de Spring Boot

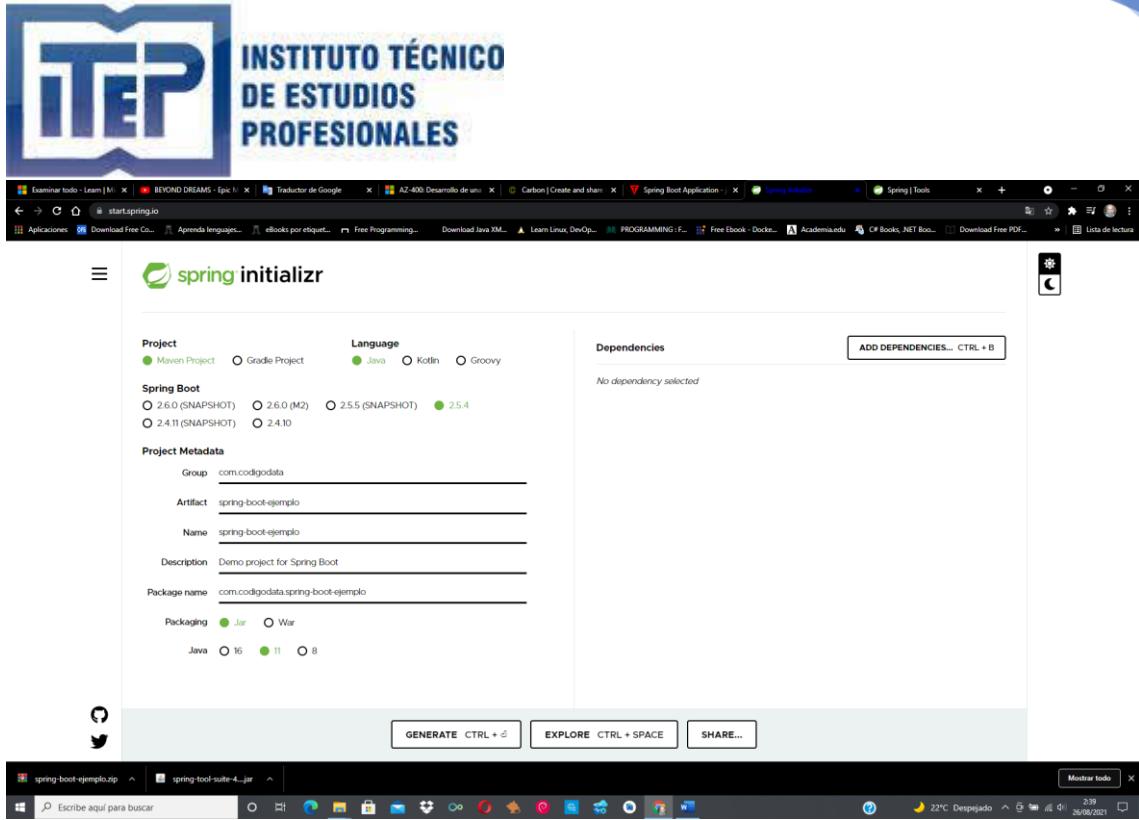
Los siguientes son los pasos para crear un proyecto Spring Boot simple.

**Paso 1:** Abra Spring initializr <https://start.spring.io> .

**Paso 2:** proporcione el nombre del **grupo** y del **artefacto** . Hemos proporcionado el nombre de grupo **com.javatpoint** y Artifact **spring-boot-example** .

**Paso 3:** Ahora haga clic en el botón **Generar** .





Cuando hacemos clic en el botón Generar, comienza a empaquetar el proyecto en un archivo **.rar** y descarga el proyecto.

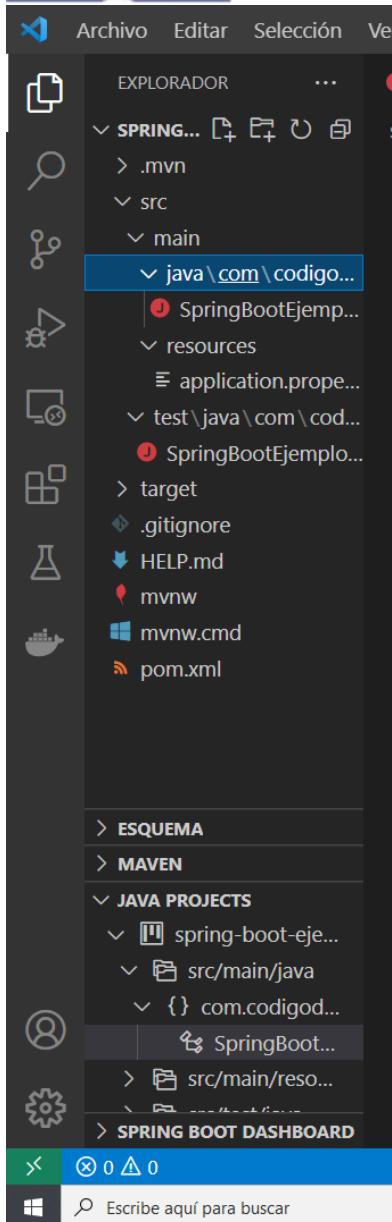
**Paso 4:** Extraiga el archivo **RAR**.

**Paso 5: importa** la carpeta.

Archivo -> Importar -> Proyecto Maven existente -> Siguiente -> Examinar -> Seleccione el proyecto -> Finalizar

Se necesita algún tiempo para importar el proyecto. Cuando el proyecto se importa correctamente, podemos ver el directorio del proyecto en el **Explorador de paquetes**. La siguiente imagen muestra el directorio del proyecto:





## SpringBootExampleApplication.java





INSTITUTO TÉCNICO  
DE ESTUDIOS  
PROFESIONALES

```
package com.codigodata.springbootejemplo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootEjemploApplication {

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

}
```

### pom.xml





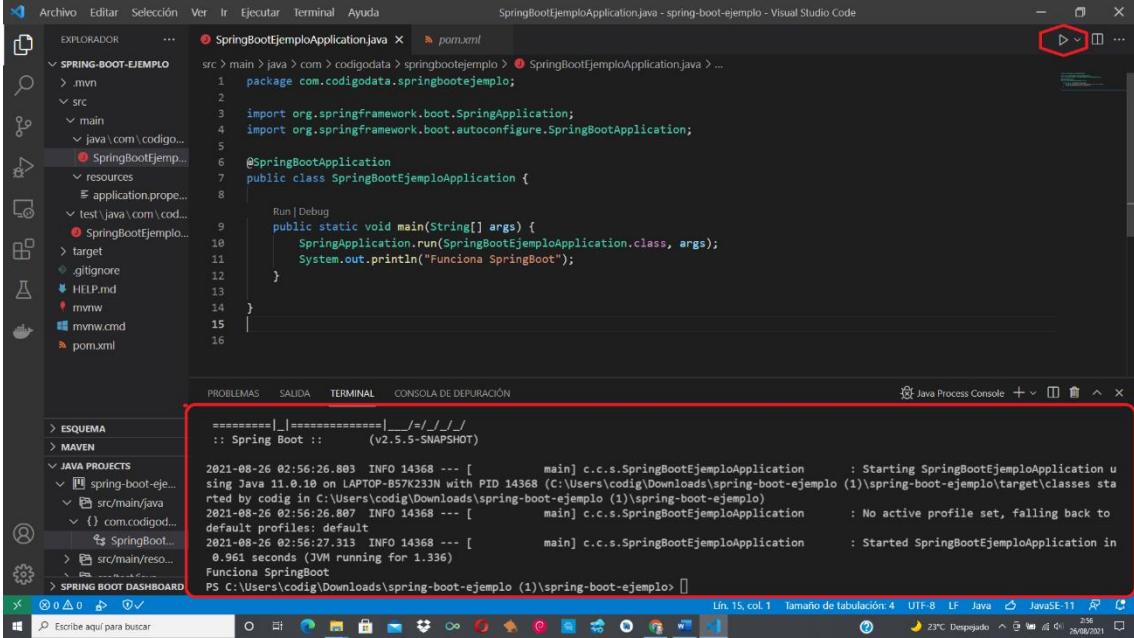
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.5-SNAPSHOT</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.codigodata</groupId>
  <artifactId>spring-boot-ejemplo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>spring-boot-ejemplo</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>11</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>

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

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
  <repositories>
    <repository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
    <repository>
      <id>spring-snapshots</id>
      <name>Spring Snapshots</name>
      <url>https://repo.spring.io/snapshot</url>
      <releases>
        <enabled>false</enabled>
      </releases>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
    <pluginRepository>
      <id>spring-snapshots</id>
      <name>Spring Snapshots</name>
      <url>https://repo.spring.io/snapshot</url>
      <releases>
        <enabled>false</enabled>
      </releases>
    </pluginRepository>
  </pluginRepositories>
</project>
```



### Paso 6: Ejecute el archivo **SpringBootExampleApplication.java**.



```

  Archivo Editar Selección Ver Ir Ejecutar Terminal Ayuda SpringBootEjemploApplication.java - spring-boot-ejemplo - Visual Studio Code
  EXPLORADOR ... SpringBootEjemploApplication.java x pom.xml
  SPRING-BOOT-EJEMPLO
  > .mvn
  > src
  > main
  > java com.codigodata...
  > com.codigodata...
  > SpringBootEjemplo...
  > resources
  > application.prop...
  > test/java/com/cod...
  > target
  > .gitignore
  > HELP.md
  > mvnw
  > mvnw.cmd
  > pom.xml

  src/main/java/com/codigodata/springbootejemplo/SpringBootEjemploApplication.java
  package com.codigodata.springbootejemplo;
  import org.springframework.boot.SpringApplication;
  import org.springframework.boot.autoconfigure.SpringBootApplication;
  @SpringBootApplication
  public class SpringBootEjemploApplication {
    public static void main(String[] args) {
      SpringApplication.run(SpringBootEjemploApplication.class, args);
      System.out.println("Funciona SpringBoot");
    }
  }

  PROBLEMAS SALIDA TERMINAL CONSOLA DE DEPURACIÓN Java Process Console + x ^ x
  ======|_|=====|_|=/_/_/
  :: Spring Boot :: (v2.5.5-SNAPSHOT)
  2021-08-26 02:56:26.803 INFO 14368 --- [           main] c.c.s.SpringBootEjemploApplication      : Starting SpringBootEjemploApplication u...
  sing Java 11.0.10 on LAPTOP-B57K23JN with PID 14368 (C:\Users\codig\Downloads\spring-boot-ejemplo (1)\spring-boot-ejemplo)
  2021-08-26 02:56:26.807 INFO 14368 --- [           main] c.c.s.SpringBootEjemploApplication      : No active profile set, falling back to ...
  default profiles: default
  2021-08-26 02:56:27.313 INFO 14368 --- [           main] c.c.s.SpringBootEjemploApplication      : Started SpringBootEjemploApplication in ...
  0.961 seconds (JVM running for 1.336)
  Funciona SpringBoot
  PS C:\Users\codig\Downloads\spring-boot-ejemplo (1)\spring-boot-ejemplo> []
  
```

## Spring Boot CLI

Es una herramienta que puede descargar desde el sitio oficial de Spring Framework. Aquí, estamos explicando los pasos.

Descargue la herramienta CLI del sitio oficial como lo estamos haciendo aquí.

## Spring Boot CLI

Es una herramienta que puede descargar desde el sitio oficial de Spring Framework. Aquí, estamos explicando los pasos.

Descargue la herramienta CLI del sitio oficial como lo estamos haciendo aquí.





**10.2.1 Manual installation**

You can download the Spring CLI distribution from the Spring software repository:

- [spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin.zip](#)
- [spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin.tar.gz](#)

Cutting edge snapshot distributions are also available.

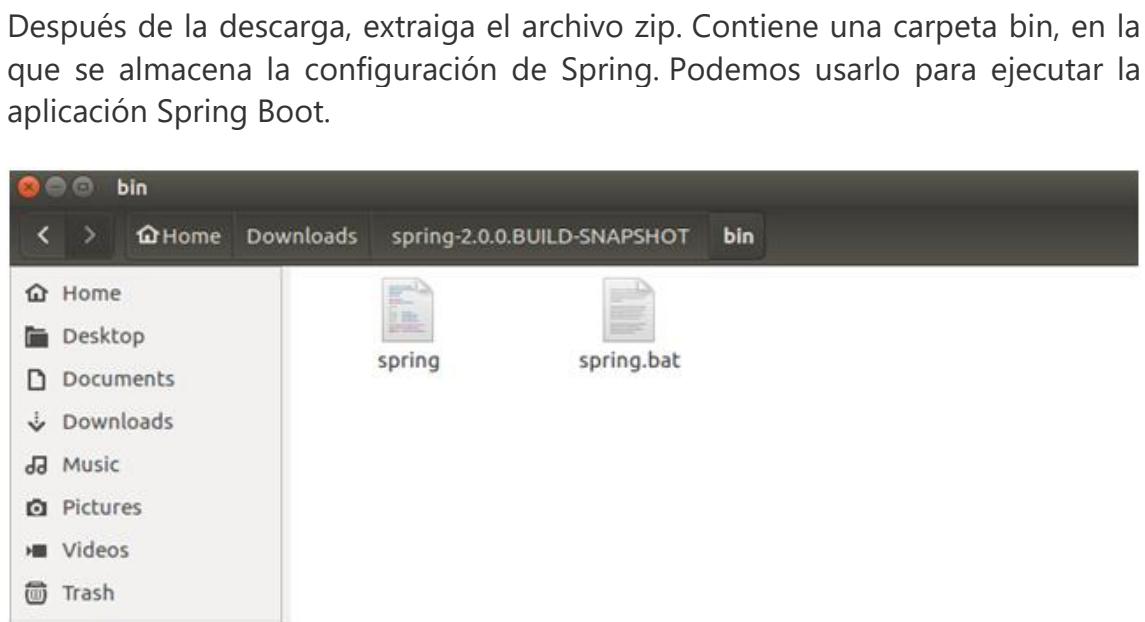
Once downloaded, follow the `INSTALL.txt` instructions from the directory in the `.zip` file, or alternatively you can use `java`.

**10.2.2 Installation with SDKMAN!**

SDKMAN! (The Software Development Kit Manager) can be used with SDKMAN! from [sdkman.io](#) and install Spring Boot with

```
$ sdk install springboot
$ spring --version
Spring Boot v2.0.0.BUILD-SNAPSHOT
```

If you are developing features for the CLI and want easy access to the version you just built, follow these extra instructions.



Después de la descarga, extraiga el archivo zip. Contiene una carpeta bin, en la que se almacena la configuración de Spring. Podemos usarlo para ejecutar la aplicación Spring Boot.

Abra el terminal y cd en la ubicación bin de la carpeta cli.





```
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin
root@irfan-GB-BXBT-2807: /home/irfan# cd Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin/
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin#
```

Crea un archivo maravilloso.

```
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin
root@irfan-GB-BXBT-2807: /home/irfan# cd Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin/
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin# vi SpringBootCliExample.groovy
```

Crea un controlador en el archivo maravilloso.

```
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin
@RestController
class SpringBootCliExample{

    @RequestMapping("/cli-example")
    String index() {
        "<h1>JavaTpoint Greets!</h1> <br>Spring Boot Example by using CLI is running successfully."
    }
}
```

Ejecuta este archivo

Usando el siguiente comando.

1. ./spring ejecutar SpringBootCliExample.groovy

```
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin
root@irfan-GB-BXBT-2807: /home/irfan# cd Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin/
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin# vi SpringBootCliExample.groovy
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin# ./spring run SpringBootCliExample.groovy
```

Después de ejecutar el comando anterior, comienza la ejecución y produce el siguiente resultado.





```
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin
:: Spring Boot :: (v2.0.0.BUILD-SNAPSHOT)

2017-03-26 18:41:27.222  INFO 14985 --- [    runner-0] o.s.boot.SpringApplication      : Starting application on irfan-GB-BXBT-2807 with PID 14985 (started by root in /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin)
2017-03-26 18:41:27.244  INFO 14985 --- [    runner-0] o.s.boot.SpringApplication      : No active profile set, falling back to default profiles: default
2017-03-26 18:41:28.991  INFO 14985 --- [    runner-0] ConfigServletWebServerApplicationContext : Refreshing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@bb9a50d: startup date [Sun Mar 26 18:41:28 IST 2017]; root of context hierarchy
2017-03-26 18:41:35.569  INFO 14985 --- [    runner-0] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2017-03-26 18:41:35.637  INFO 14985 --- [    runner-0] o.apache.catalina.core.StandardService : Starting service Tomcat
2017-03-26 18:41:35.640  INFO 14985 --- [    runner-0] org.apache.catalina.core.StandardEng
```

Y después de muchas líneas. Muestra el estado actual de la aplicación de la siguiente manera.

```
root@irfan-GB-BXBT-2807: /home/irfan/Downloads/spring-2.0.0.BUILD-SNAPSHOT/bin
2017-03-26 18:41:39.240  INFO 14985 --- [    runner-0] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[{[error]},produces=[text/html]]" onto public org.springframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
2017-03-26 18:41:39.241  INFO 14985 --- [    runner-0] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[{[error]}]" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController.error(javax.servlet.http.HttpServletRequest)
2017-03-26 18:41:39.403  INFO 14985 --- [    runner-0] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-03-26 18:41:39.405  INFO 14985 --- [    runner-0] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.resource.ResourceHttpRequestHandler]
2017-03-26 18:41:39.609  INFO 14985 --- [    runner-0] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-03-26 18:41:42.093  INFO 14985 --- [    runner-0] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
2017-03-26 18:41:42.329  INFO 14985 --- [    runner-0] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http)
2017-03-26 18:41:42.376  INFO 14985 --- [    runner-0] o.s.boot.SpringApplication      : Started application in 17.227 seconds (JVM running for 28.401)
```

Este proyecto se está ejecutando en el puerto 8080. Entonces, podemos invocarlo en cualquier navegador usando la siguiente URL.

1. localhost: 8080: / cli-example

Producirá la siguiente salida.





## JavaTpoint Greets!

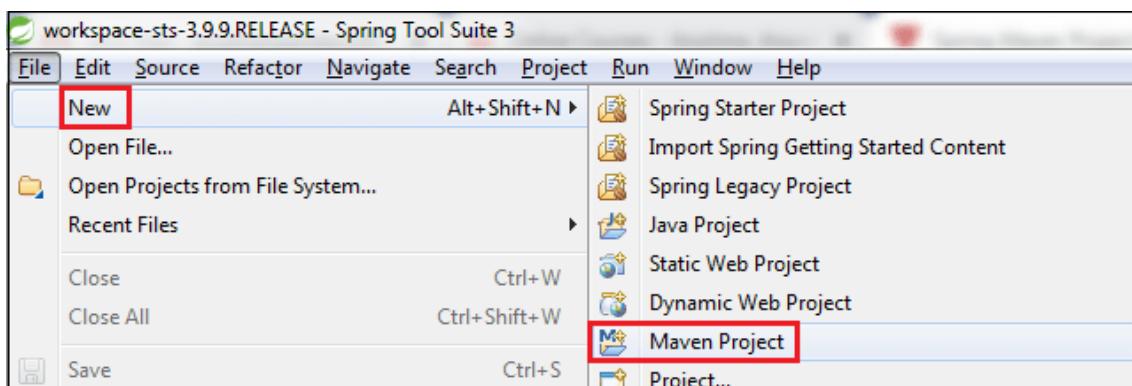
Spring Boot Example by using CLI is running successfully.

## Creación de un proyecto de Spring Boot usando STS

También podemos usar Spring Tool Suite para crear un proyecto Spring. En esta sección, crearemos un **proyecto Maven** usando **STS**.

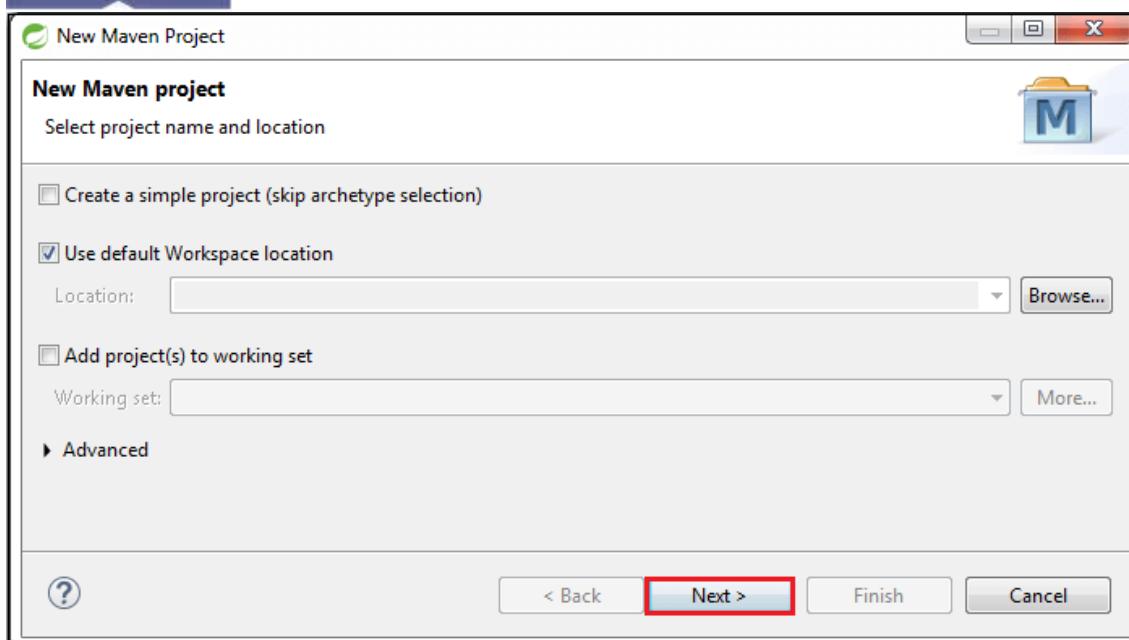
**Paso 1:** Abra Spring Tool Suite.

**Paso 2:** haga clic en el menú Archivo -> Nuevo -> Proyecto Maven



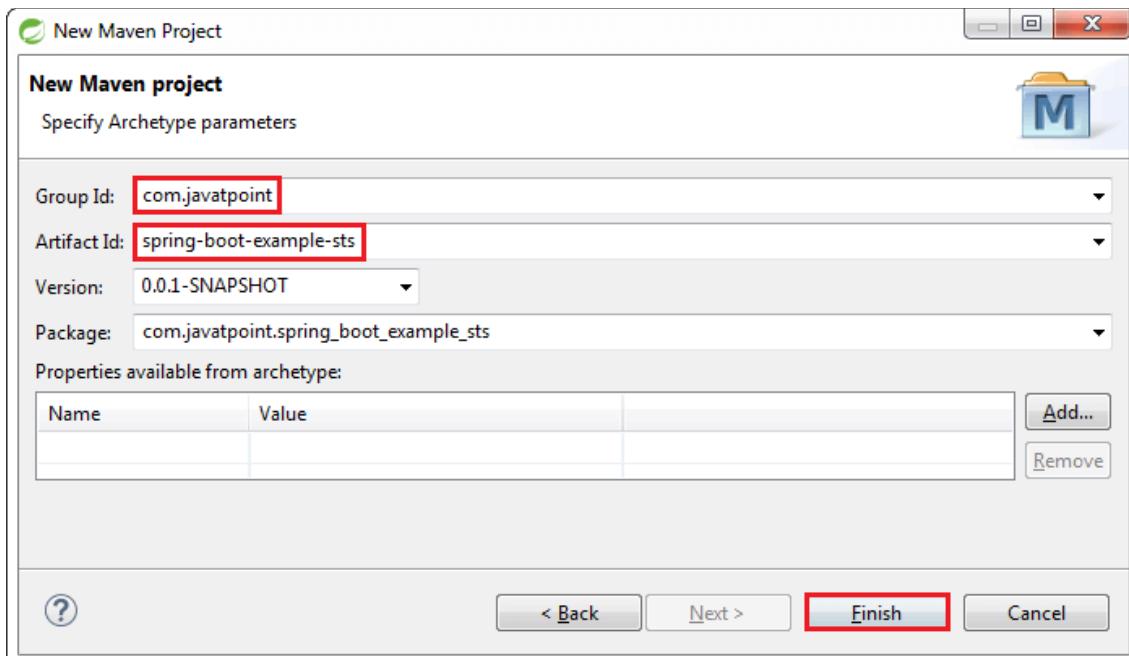
Muestra el asistente de New Maven Project. Haga clic en el botón **Siguiente**.



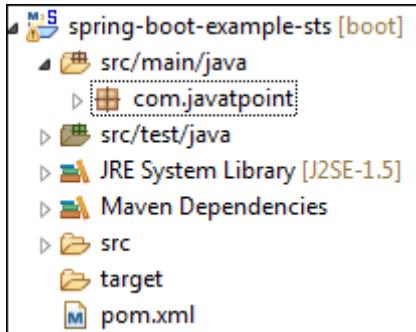


**Paso 3:** Seleccione el **inicio rápido de maven-archetype** y haga clic en el botón **Siguiente**.

**Paso 4:** proporcione el **Id. De grupo** y el **Id. De artefacto**. Hemos proporcionado el Id. De grupo **com.javatpoint** y el Id. De artefacto **spring-boot-example-sts**. Ahora haga clic en el botón **Finalizar**.



Cuando hacemos clic en el botón Finalizar, se crea el directorio del proyecto, como se muestra en la siguiente imagen.



**Paso 5:** Abra el archivo **App.java**. Encontramos el siguiente código que está por defecto.

### App.java



```
package com.javatpoint;
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

El proyecto Maven tiene un archivo **pom.xml** que contiene la siguiente configuración predeterminada.

### pom.xml





```
● ● ●  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
<modelVersion>4.0.0</modelVersion>  
<groupId>com.javatpoint</groupId>  
<artifactId>spring-boot-example-sts</artifactId>  
<version>0.0.1-SNAPSHOT</version>  
<packaging>jar</packaging>  
<name>spring-boot-example-sts</name>  
<url>http://maven.apache.org</url>  
<properties>  
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
</properties>  
<dependencies>  
<dependency>  
<groupId>junit</groupId>  
<artifactId>junit</artifactId>  
<version>3.8.1</version>  
<scope>test</scope>  
</dependency>  
</dependencies>  
</project>
```

**Paso 6:** agregue la versión de Java dentro de la etiqueta <properties> .

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

**Paso 7:** Para hacer un proyecto Spring Boot, necesitamos configurarlo. Por lo tanto, estamos agregando la dependencia **principal del arrancador de arranque de primavera** en el archivo **pom.xml** . Parent se utiliza para declarar que nuestro proyecto es un hijo de este proyecto principal.





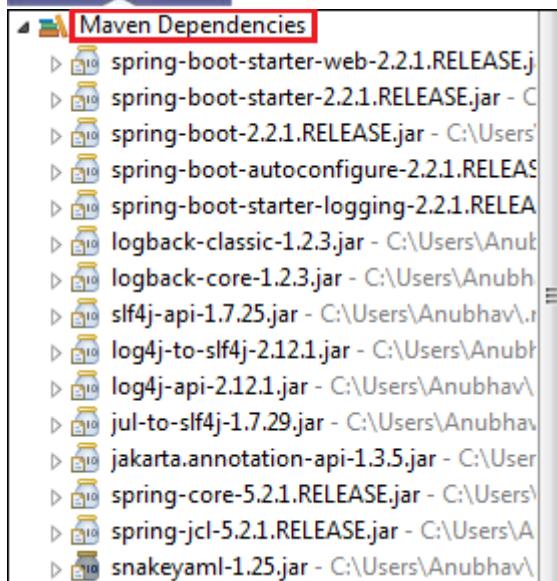
```
● ● ●  
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-parent</artifactId>  
  <version>2.2.1.RELEASE</version>  
  <type>pom</type>  
</dependency>
```

**Paso 8:** agregue la dependencia **spring-boot-starter-web** en el archivo **pom.xml**.

```
● ● ●  
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-web</artifactId>  
  <version>2.2.1.RELEASE</version>  
</dependency>
```

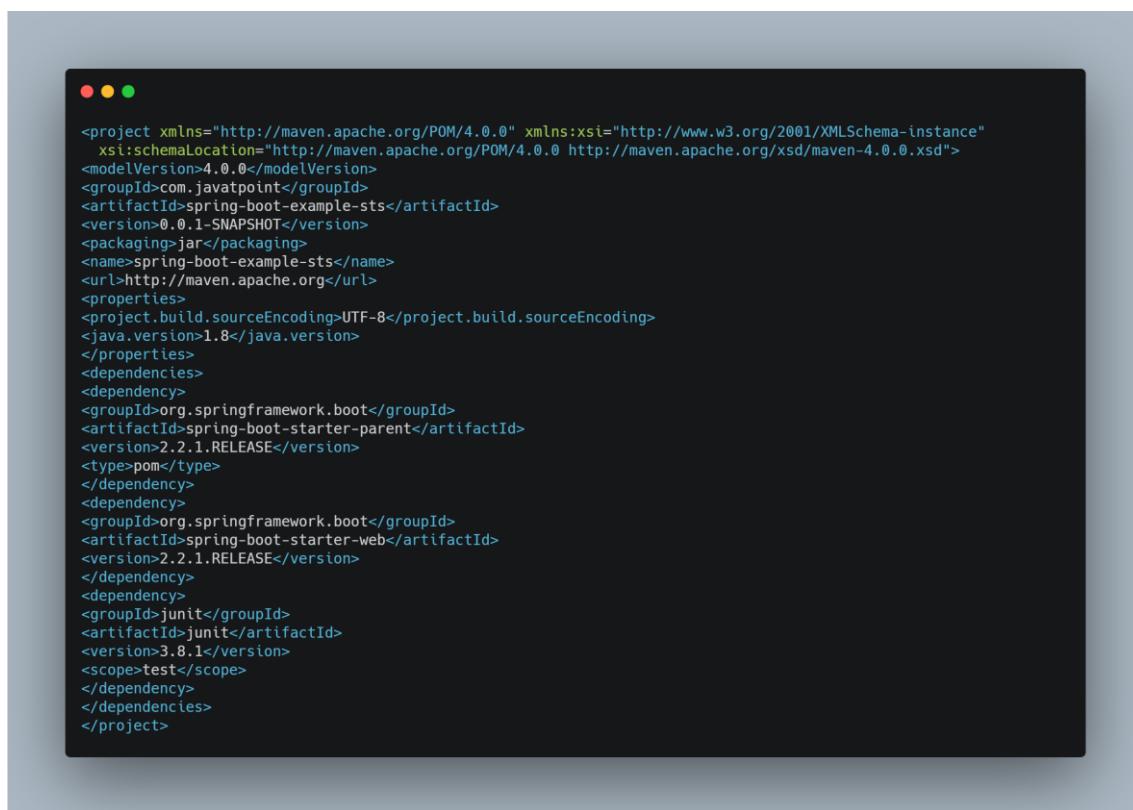
**Nota:** Cuando agregamos las dependencias en el archivo pom, descarga el archivo jar relacionado. Podemos ver los archivos jar descargados en la carpeta Maven Dependencies del directorio del proyecto.





Después de agregar todas las dependencias, el archivo pom.xml tiene el siguiente aspecto:

#### pom.xml



```

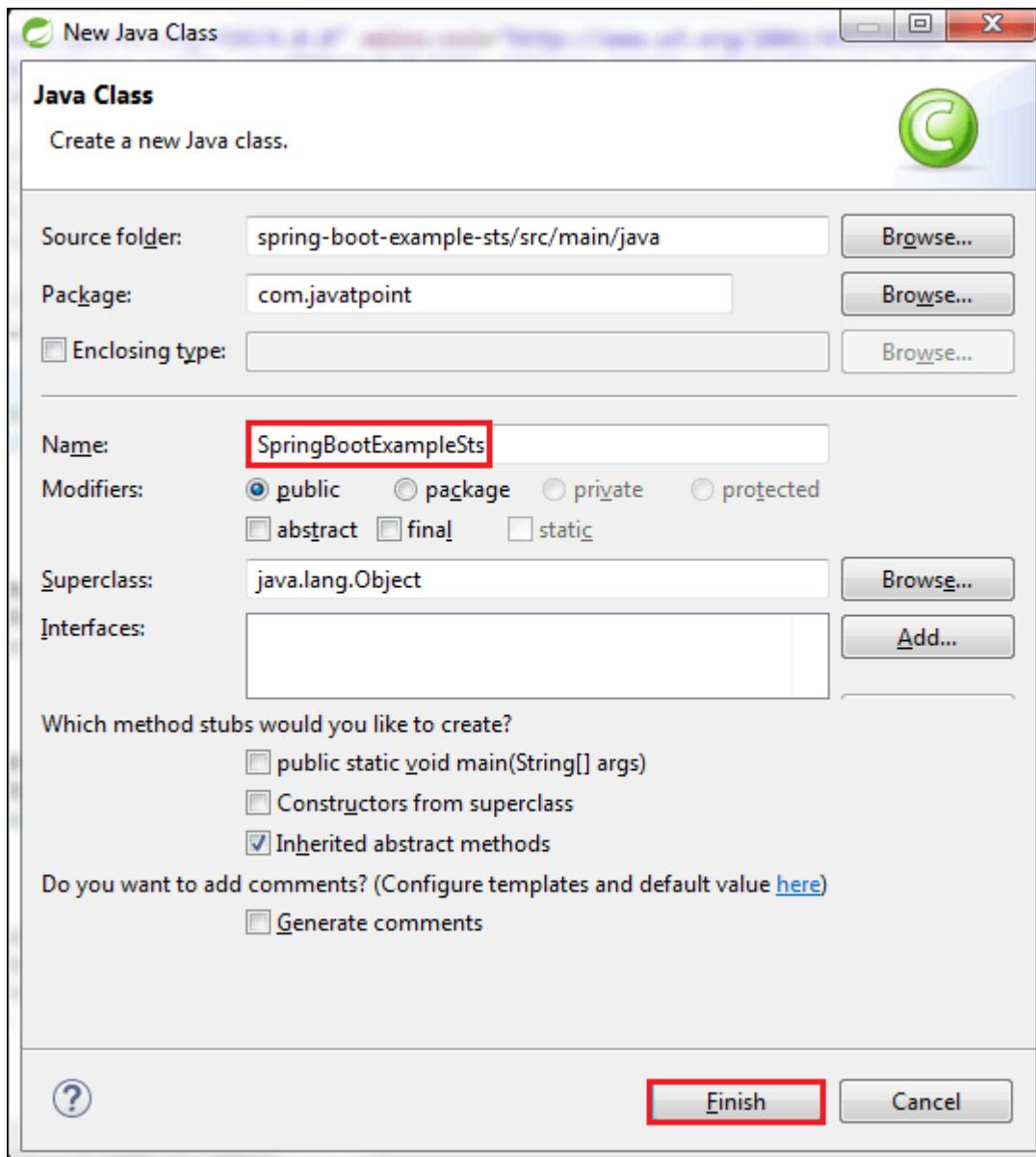
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatpoint</groupId>
  <artifactId>spring-boot-example-sts</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>spring-boot-example-sts</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.2.1.RELEASE</version>
      <type>pom</type>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>2.2.1.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

**Paso 9:** Cree una clase con el nombre **SpringBootExampleSts** en el paquete **com.javatpoint**.



Haga clic derecho en el nombre del paquete -> Nuevo -> Clase -> proporcione el nombre de la clase -> Finalizar



**Paso 10:** Despu s de crear el archivo de clase, llame al m todo est tico **run ()** de la clase SpringApplication. En el siguiente c digo, llamamos al m todo run () y pasamos el nombre de la clase como argumento.





```
SpringApplication.run(SpringBootExampleSts.class, args);
```

**Paso 11:** Anote la clase agregando una anotación **@SpringBootApplication** .

### **@SpringBootApplication**

Se utiliza una sola anotación @SpringBootApplication para habilitar las siguientes anotaciones:

- **@EnableAutoConfiguration:** Habilita el mecanismo de configuración automática de Spring Boot.
- **@ComponentScan:** escanea el paquete donde se encuentra la aplicación.
- **@Configuration:** Nos permite registrar beans extra en el contexto o importar clases de configuración adicionales.

### **SpringBootApplicationSts.java**



```
package com.javatpoint;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class SpringBootExampleSts
{
    public static void main(String[] args)
    {
        SpringApplication.run(SpringBootExampleSts.class, args);
    }
}
```

**Paso:** Ejecute el archivo **SpringBootExampleSts.java** , como aplicación Java. Muestra lo siguiente en la consola.





La línea **Started SpringBootExampleSts en 5.038 segundos (JVM ejecutándose para 6.854)** en la consola muestra que la aplicación está en funcionamiento.

# Anotaciones de Spring Boot

Spring Boot Annotations es una forma de metadatos que proporciona datos sobre un programa. En otras palabras, las anotaciones se utilizan para proporcionar información **complementaria** sobre un programa. No es parte de la aplicación que desarrollamos. No tiene un efecto directo sobre el funcionamiento del código que anotan. No cambia la acción del programa compilado.

En esta sección, vamos a discutir algunas **anotaciones de Spring Boot** importantes que usaremos más adelante en este tutorial.

# Anotaciones de Core Spring Framework

**@Required:** Se aplica al método de establecimiento de **beans**. Indica que el bean anotado debe **llenarse** en el momento de la configuración con la propiedad requerida; de lo contrario, lanza una excepción **BeanInitializationException**.

## Ejemplo





```
public class Machine
{
    private Integer cost;
    @Required
    public void setCost(Integer cost)
    {
        this.cost = cost;
    }
    public Integer getCost()
    {
        return cost;
    }
}
```

**@Autowired:** Spring proporciona cableado automático basado en anotaciones al proporcionar una anotación @Autowired. Se utiliza para conectar automáticamente el bean de Spring en los métodos setter, la variable de instancia y el constructor. Cuando usamos la anotación @Autowired, el contenedor de resorte conecta automáticamente el bean al hacer coincidir el tipo de datos.

### Ejemplo





```
● ● ●  
  
@Component  
public class Customer  
{  
    private Person person;  
    @Autowired  
    public Customer(Person person)  
    {  
        this.person=person;  
    }  
}
```

**@Configuration:** es una anotación a nivel de clase. La clase anotada con @Configuration utilizada por Spring Containers como fuente de definiciones de beans.

### Ejemplo





```
● ● ●  
  
@Configuration  
public class Vehicle  
{  
    @BeanVehicle engine()  
    {  
        return new Vehicle();  
    }  
}
```

**@ComponentScan:** Se usa cuando queremos escanear un paquete en busca de beans. Se usa con la anotación @Configuration. También podemos especificar los paquetes base para buscar componentes Spring.

### Ejemplo

```
● ● ●  
  
@ComponentScan(basePackages = "com.javatpoint")  
@Configuration  
public class ScanComponent  
{  
    // ...  
}
```

**@Bean:** es una anotación a nivel de método. Es una alternativa a la etiqueta XML <bean>. Le dice al método que produzca un bean para que sea administrado por Spring Container.





## Ejemplo

```
● ● ●  
  
@Bean  
public BeanExample beanExample()  
{  
    return new BeanExample ();  
}
```

## Anotaciones de estereotipo de Spring Framework

**@Component:** es una anotación a nivel de clase. Se utiliza para marcar una clase Java como bean. Se encuentra una clase Java anotada con **@Component** durante la ruta de clase . Spring Framework lo recoge y lo configura en el contexto de la aplicación como **Spring Bean** .

## Ejemplo

```
● ● ●  
  
@Component  
public class Student  
{  
    .....  
}
```

**@Controller:** @Controller es una anotación a nivel de clase. Es una especialización de **@Component** . Marca una clase como manejador de solicitudes web. A menudo se utiliza para servir páginas web. De forma



predeterminada, devuelve una cadena que indica qué ruta redireccionar. Se usa principalmente con la anotación **@RequestMapping**.

### Ejemplo

```
● ● ●

@Controller
@RequestMapping("books")
public class BooksController
{
    @RequestMapping(value = "/{name}", method = RequestMethod.GET)
    public Employee getBooksByName()
    {
        return booksTemplate;
    }
}
```

**@Service:** también se usa a nivel de clase. Le dice a Spring que la clase contiene la **lógica empresarial**.

### Ejemplo

```
● ● ●

package com.javatpoint;
@Service
public class TestService
{
    public void service1()
    {
        //business code
    }
}
```

**@Repository:** es una anotación a nivel de clase. El repositorio es un **DAO** (objeto de acceso a datos) que accede a la base de datos directamente. El repositorio realiza todas las operaciones relacionadas con la base de datos.





● ● ●

```
package com.javatpoint;
@Repository
public class TestRepository
{
    public void delete()
    {
        //persistence code
    }
}
```

## Anotaciones de Spring Boot

- **@EnableAutoConfiguration:** configura automáticamente el bean que está presente en la ruta de clase y lo configura para ejecutar los métodos. El uso de esta anotación se reduce en la versión Spring Boot 1.2.0 porque los desarrolladores proporcionaron una alternativa de la anotación, es decir, **@SpringBootApplication** .
- **@SpringBootApplication:** es una combinación de tres anotaciones **@EnableAutoConfiguration**, **@ComponentScan** y **@Configuration** .

## Spring MVC y anotaciones REST

- **@RequestMapping:** se utiliza para mapear las **solicitudes web** . Tiene muchos elementos opcionales como **consumes**, **header**, **method**, **name**, **params**, **path**, **produces** y **value** . Lo usamos tanto con la clase como con el método.

## Ejemplo





```
● ● ●  
  
@Controller  
public class BooksController  
{  
    @RequestMapping("/computer-science/books")  
    public String getAllBooks(Model model)  
    {  
        //application code  
        return "bookList";  
    }  
}
```

- **@GetMapping:** asigna las solicitudes **HTTP GET** en el método de controlador específico. Se utiliza para crear un endpoint de servicio web que **recupera**. Se utiliza en lugar de utilizar: **@RequestMapping (método = RequestMethod.GET)**
- **@PostMapping:** asigna las solicitudes **HTTP POST** en el método de controlador específico. Se usa para crear un endpoint de servicio web que **crea**. Se usa en lugar de usar: **@RequestMapping (método = RequestMethod.POST)**
- **@PutMapping:** asigna las solicitudes **HTTP PUT** en el método de controlador específico. Se usa para crear un endpoint de servicio web que **crea o actualiza**. Se usa en lugar de usar: **@RequestMapping (método = RequestMethod.PUT)**
- **@DeleteMapping:** asigna las solicitudes **HTTP DELETE** en el método de controlador específico. Se utiliza para crear un endpoint de servicio web que **elimina** un recurso. Se usa en lugar de usar: **@RequestMapping (método = RequestMethod.DELETE)**
- **@PatchMapping:** asigna las solicitudes **HTTP PATCH** en el método de controlador específico. Se usa en lugar de usar: **@RequestMapping (método = RequestMethod.PATCH)**





- **@RequestBody:** se utiliza para **vincular una** solicitud HTTP con un objeto en un parámetro de método. Internamente, utiliza **HTTP MessageConverters** para convertir el cuerpo de la solicitud. Cuando anotamos un parámetro de método con **@RequestBody**, el marco de Spring vincula el cuerpo de la solicitud HTTP entrante a ese parámetro.
- **@ResponseBody:** **Vincula** el valor de retorno del método al cuerpo de la respuesta. Le dice a Spring Boot Framework que serialice una devolución de un objeto en formato JSON y XML.
- **@PathVariable:** se utiliza para extraer los valores del URI. Es más adecuado para el servicio web RESTful, donde la URL contiene una variable de ruta. Podemos definir múltiples @PathVariable en un método.
- **@RequestParam:** se utiliza para extraer los parámetros de consulta de la URL. También se conoce como **query parameter (parámetro de consulta)**. Es más adecuado para aplicaciones web. Puede especificar valores predeterminados si el parámetro de consulta no está presente en la URL.
- **@RequestHeader:** se usa para obtener los detalles sobre los encabezados de solicitud HTTP. Usamos esta anotación como **parámetro de método**. Los elementos opcionales de la anotación son **name, required, value, defaultValue**. Para cada detalle del encabezado, debemos especificar anotaciones separadas. Podemos usarlo varias veces en un método.
- **@RestController:** se puede considerar como una combinación de las anotaciones **@Controller** y **@ResponseBody**. La anotación **@RestController** se anota en sí misma con la anotación **@ResponseBody**. Elimina la necesidad de anotar cada método con **@ResponseBody**.
- **@RequestAttribute:** **Vincula** un parámetro de método a un atributo de solicitud. Proporciona un acceso conveniente a los atributos de solicitud desde un método de controlador. Con la ayuda de la anotación **@RequestAttribute**, podemos acceder a los objetos que se rellenan en el lado del servidor.





## Gestión de dependencias de Spring Boot

Spring Boot gestiona las dependencias y la configuración de forma automática. Cada versión de Spring Boot proporciona una lista de dependencias que admite. La lista de dependencias está disponible como parte de las **listas de materiales** (spring-boot-dependencies) que se pueden usar con **Maven**. Entonces, no necesitamos especificar la versión de las dependencias en nuestra configuración. Spring Boot se maneja solo. Spring Boot actualiza todas las dependencias automáticamente de manera consistente cuando actualizamos la versión de Spring Boot.

### Ventajas de la gestión de dependencias

- Proporciona la centralización de la información de dependencia especificando la versión de Spring Boot en un solo lugar. Ayuda cuando cambiamos de una versión a otra.
- Evita la discordancia de las diferentes versiones de las bibliotecas Spring Boot.
- Solo necesitamos escribir un nombre de biblioteca especificando la versión. Es útil en proyectos de varios módulos.

**Nota:** Spring Boot también permite anular la versión de las dependencias, si es necesario.

### Sistema de gestión de dependencias de Maven

El proyecto Maven hereda las siguientes características de **spring-boot-starter-parent**:

- La **versión** predeterminada del **compilador de Java**
- **Codificación de fuente UTF-8**
- Hereda una **sección de dependencia** de spring-boot-dependency-pom. Gestiona la versión de dependencias comunes. Ignora la etiqueta **<version>** para esas dependencias.
- Dependencias, heredadas del POM spring-boot-dependencies
- **Filtrado de recursos** sensato
- **Configuración de complementos** sensible





## Heredar padre inicial

El siguiente **spring-boot-starter-parent** se hereda automáticamente cuando configuramos el proyecto.

```
● ● ●

<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.2.BUILD-SNAPSHOT</version>      <!-- lookup parent from repository -->
<relativePath/>
</parent>
```

Por ejemplo, si queremos agregar otra dependencia con el mismo artefacto que ya hemos injectado, inyecte esa dependencia nuevamente dentro de la etiqueta **<properties>** para anular la anterior.

## Cambiar la versión de Java

También podemos cambiar la versión de Java usando la etiqueta **<java.version>**.

```
● ● ●

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

Adición del complemento Spring Boot Maven

También podemos **agregar el complemento Maven** en nuestro archivo **pom.xml**. Envuelve el proyecto en un archivo **jar** ejecutable .





```
● ● ●

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

## Spring Boot sin Parent POM

Si no queremos usar la dependencia **spring-boot starter-parent**, pero aún queremos aprovechar la administración de dependencias, podemos usar la etiqueta **<scope>**, de la siguiente manera:

```
● ● ●

<><dependencyManagement>
<dependencies>
<dependency><!-- Import dependency management from Spring Boot -->
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>2.2.2.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

La dependencia anterior no permite anular. Para lograr la anulación, necesitamos agregar una entrada dentro de la etiqueta **<dependencyManagement>** de nuestro proyecto antes de la entrada `spring-boot-dependencies`.

Por ejemplo, para actualizar otro **spring-data-releasetrain**, agregue la siguiente dependencia en el archivo pom.xml.



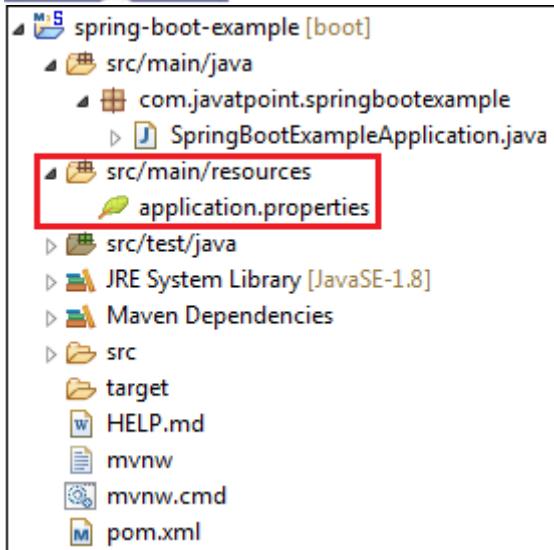
```
● ● ●

<dependencyManagement>
<dependencies>
<!--Override Spring Data release train-->
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-releasetrain</artifactId>
<version>Fowler-SR2</version>
<type>pom</type>
<scope>import</scope>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>2.2.2.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

## Propiedades de la aplicación Spring Boot

Spring Boot Framework viene con un mecanismo integrado para la configuración de la aplicación mediante un archivo llamado **application.properties**. Se encuentra dentro de la carpeta **src / main / resources**, como se muestra en la siguiente figura.





Spring Boot proporciona varias propiedades que se pueden configurar en el archivo **application.properties**. Las propiedades tienen valores predeterminados. Podemos establecer una propiedad (s) para la aplicación Spring Boot. Spring Boot también nos permite definir nuestra propia propiedad si es necesario.

El archivo `application.properties` nos permite ejecutar una aplicación en un **entorno diferente**. En resumen, podemos usar el archivo `application.properties` para:

- Configurar el marco de Spring Boot
- definir las propiedades de configuración personalizadas de nuestra aplicación





## Ejemplo de application.properties



```
#configuring application name
spring.application.name = demoApplication
#configuring port
server.port = 8081
```

En el ejemplo anterior, hemos configurado el **nombre** y el **puerto de la aplicación**. El puerto 8081 indica que la aplicación se ejecuta en el puerto **8081**.

**Nota:** Las líneas que comienzan con # son comentarios.

### Archivo de propiedades YAML

Spring Boot proporciona otro archivo para configurar las propiedades que se llama archivo **yml**. El archivo Yaml funciona porque el jar de **Snake YAML** está presente en la ruta de clase. En lugar de usar el archivo application.properties, también podemos usar el archivo application.yml, pero el archivo **Yml** debe estar presente en la ruta de clases.

### Ejemplo de application.yml





```
spring:  
application:  
name: demoApplication  
server:  
port: 8081
```

En el ejemplo anterior, hemos configurado el **nombre** y el **puerto de la aplicación**. El puerto 8081 indica que la aplicación se ejecuta en el puerto **8081**.

## Categorías de propiedades de Spring Boot

Hay **dieciséis** categorías de propiedades de Spring Boot que son las siguientes:

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

## Tabla de propiedades de la aplicación

Las siguientes tablas proporcionan una lista de propiedades comunes de Spring Boot:

Propiedad	Valores predeterminados	Descripción
<b>Debug</b>	false	Habilita los registros de depuración.
<b>spring.application.name</b>		Se utiliza para establecer el nombre de la aplicación.
<b>spring.application.admin.enabled</b>	false	Se utiliza para habilitar las funciones de administración de la aplicación.
<b>spring.config.name</b>	application	Se utiliza para establecer el nombre del archivo de configuración.
<b>spring.config.location</b>		Se utiliza para configurar el nombre del archivo.
<b>server.port</b>	8080	Configura el puerto del servidor HTTP
<b>server.servlet.context-path</b>		Configura la ruta de contexto de la aplicación.





<b>logging.file.path</b>		Configura la ubicación del archivo de registro.
<b>spring.banner.charset</b>	UTF-8	Codificación de archivos de banner.
<b>spring.banner.location</b>	classpath: banner.txt	Se utiliza para establecer la ubicación del archivo de banner.
<b>logging.file</b>		Se utiliza para establecer el nombre del archivo de registro. Por ejemplo, data.log.
<b>spring.application.index</b>		Se utiliza para establecer el índice de la aplicación.
<b>spring.application.name</b>		Se utiliza para establecer el nombre de la aplicación.
<b>spring.application.admin.enabled</b>	false	Se utiliza para habilitar las funciones de administración de la aplicación.
<b>spring.config.location</b>		Se utiliza para configurar las ubicaciones de los archivos.
<b>spring.config.name</b>	application	Se utiliza para configurar el nombre del archivo.





<b>spring.mail.default-encoding</b>	UTF-8	Se utiliza para establecer la codificación predeterminada de MimeMessage.
<b>spring.mail.host</b>		Se utiliza para configurar el host del servidor SMTP. Por ejemplo, smtp.example.com.
<b>spring.mail.password</b>		Se utiliza para establecer la contraseña de inicio de sesión del servidor SMTP.
<b>spring.mail.port</b>		Se utiliza para configurar el puerto del servidor SMTP.
<b>spring.mail.test-connection</b>	false	Se utiliza para probar que el servidor de correo está disponible al iniciarse.
<b>spring.mail.username</b>		Se utiliza para configurar el usuario de inicio de sesión del servidor SMTP.
<b>spring.main.sources</b>		Se utiliza para establecer fuentes para la aplicación.
<b>server.address</b>		Se utiliza para establecer la





		dirección de red a la que debe vincularse el servidor.
<b>server.connection-timeout</b>		Se utiliza para establecer el tiempo en milisegundos que los conectores esperarán otra solicitud HTTP antes de cerrar la conexión.
<b>server.context-path</b>		Se utiliza para establecer la ruta de contexto de la aplicación.
<b>server.port</b>	8080	Se utiliza para configurar el puerto HTTP.
<b>server.server-header</b>		Se utiliza para el encabezado de respuesta del servidor (no se envía ningún encabezado si está vacío)
<b>server.servlet-path</b>	/	Se utiliza para establecer la ruta del servlet del despachador principal.
<b>server.ssl.enabled</b>		Se utiliza para habilitar el soporte SSL.
<b>spring.http.multipart.enabled</b>	true	Se utiliza para habilitar la compatibilidad con cargas de varias partes.





<b>spring.servlet.multipart.max-file-size</b>	1 MB	Se utiliza para establecer el tamaño máximo de archivo.
<b>spring.mvc.async.request-timeout</b>		Se utiliza para configurar el tiempo en milisegundos.
<b>spring.mvc.date-format</b>		Se utiliza para configurar el formato de la fecha. Por ejemplo, dd / MM / aaaa.
<b>spring.mvc.locale</b>		Se utiliza para establecer la configuración regional de la aplicación.
<b>spring.social.facebook.app-id</b>		Se utiliza para configurar el ID de la aplicación de Facebook de la aplicación.
<b>spring.social.linkedin.app-id</b>		Se utiliza para configurar el ID de la aplicación de LinkedIn de la aplicación.
<b>spring.social.twitter.app-id</b>		Se utiliza para configurar el ID de la aplicación de Twitter de la aplicación.
<b>security.basic.authorize-mode</b>	role	Se utiliza para configurar el modo de autorización de seguridad para aplicar.





<b>security.basic.enabled</b>	true	Se utiliza para habilitar la autenticación básica.
<b>Spring.test.database.replace</b>	any	Tipo de fuente de datos existente para reemplazar.
<b>Spring.test.mockmvc.print</b>	default	Opción de impresión MVC
<b>spring.freemarker.content-type</b>	text / html	Valor del tipo de contenido
<b>server.server-header</b>		Valor que se utilizará para el encabezado de respuesta del servidor.
<b>spring.security.filter.dispatcher-type</b>	async, error, request	Tipos de despachadores de cadenas de filtros de seguridad.
<b>spring.security.filter.order</b>	-100	Orden de cadena de filtros de seguridad.
<b>spring.security.oauth2.client.registration.*</b>		Registros de clientes de OAuth.
<b>spring.security.oauth2.client.provider.*</b>		Detalles del proveedor de OAuth.

## Arrancadores Spring Boot

**Spring Boot** proporciona una serie de **iniciadores** que nos permiten agregar jar en la ruta de clases. Los **arrancadores** incorporados Spring Boot hacen que el desarrollo sea más fácil y rápido. **Spring Boot Starters** son los **descriptores de dependencia**.





En Spring Boot Framework, todos los iniciadores siguen un patrón de nomenclatura similar: **spring-boot-starter-**\*, donde \* denota un tipo particular de aplicación. Por ejemplo, si queremos usar Spring y JPA para el acceso a la base de datos, necesitamos incluir la dependencia **spring-boot-starter-data-jpa** en nuestro archivo **pom.xml** del proyecto.

## Entrantes de terceros

También podemos incluir **arrancadores de terceros** en nuestro proyecto. Pero no usamos **spring-boot-starter** para incluir la dependencia de terceros. Spring-boot-starter está reservado para los artefactos oficiales de Spring Boot. El iniciador de terceros comienza con el nombre del proyecto. Por ejemplo, el nombre del proyecto de terceros es **abc**, luego el nombre de la dependencia será **abc-spring-boot-starter**.

Spring Boot Framework proporciona los siguientes iniciadores de aplicaciones en el grupo **org.springframework.boot** .

Nombre	Descripción
<b>spring-boot-starter-thymeleaf</b>	Se utiliza para crear aplicaciones web MVC utilizando vistas de Thymeleaf.
<b>spring-boot-starter-data-couchbase</b>	Se utiliza para la base de datos orientada a documentos de Couchbase y Spring Data Couchbase.
<b>spring-boot-starter-artemis</b>	Se utiliza para mensajería JMS utilizando Apache Artemis.
<b>spring-boot-starter-web-services</b>	Se utiliza para Spring Web Services.
<b>spring-boot-starter-mail</b>	Se utiliza para admitir el envío de correo electrónico de Java Mail y Spring Framework.
<b>spring-boot-starter-data-redis</b>	Se utiliza para el almacén de datos de valor-clave de Redis con Spring Data Redis y el cliente Jedis.
<b>Spring-boot-starter-web</b>	Se utiliza para crear la aplicación web, incluidas las aplicaciones RESTful que utilizan Spring MVC. Utiliza Tomcat como contenedor integrado predeterminado.
<b>Spring-boot-starter-data-gemfire</b>	Se utiliza para almacenar datos distribuidos GemFire y Spring Data GemFire.
<b>Spring-boot-starter-activemq</b>	Se utiliza en mensajería JMS utilizando Apache ActiveMQ.
<b>Spring-boot-starter-data-elasticsearch</b>	Se utiliza en el motor de búsqueda y análisis de Elasticsearch y en Spring Data Elasticsearch.
<b>spring-boot-starter-integration</b>	Se utiliza para la integración de Spring.





<b>spring-boot-starter-test</b>	Se utiliza para probar aplicaciones Spring Boot con bibliotecas, incluidas JUnit, Hamcrest y Mockito.
<b>spring-boot-starter-jdbc</b>	Se utiliza para JDBC con el grupo de conexiones Tomcat JDBC.
<b>spring-boot-starter-mobile</b>	Se utiliza para crear aplicaciones web con Spring Mobile.
<b>spring-boot-starter-validation</b>	Se utiliza para la validación de Java Bean con Hibernate Validator.
<b>spring-boot-starter-hateoas</b>	Se utiliza para construir una aplicación web RESTful basada en hipermedia con Spring MVC y Spring HATEOAS.
<b>spring-boot-starter-jersey</b>	Se utiliza para crear aplicaciones web RESTful utilizando JAX-RS y Jersey. Una alternativa a <code>spring-boot-starter-web</code> .
<b>Spring-boot-starter-data-neo4j</b>	Se utiliza para la base de datos de gráficos de Neo4j y Spring Data Neo4j.
<b>Spring-boot-starter-data-ldap</b>	Se utiliza para Spring Data LDAP.
<b>Spring-boot-starter-websocket</b>	Se utiliza para crear aplicaciones WebSocket. Utiliza el soporte WebSocket de Spring Framework.
<b>spring-boot-starter-aop</b>	Se utiliza para la programación orientada a aspectos con Spring AOP y AspectJ.
<b>spring-boot-starter-amqp</b>	Se utiliza para Spring AMQP y Rabbit MQ.
<b>Spring-boot-starter-data-cassandra</b>	Se utiliza para la base de datos distribuida de Cassandra y Spring Data Cassandra.
<b>Spring-boot-starter-social-facebook</b>	Se utiliza para Spring Social Facebook.
<b>spring-boot-starter-jta-atomikos</b>	Se utiliza para transacciones JTA utilizando Atomikos.
<b>Spring-boot-starter-security</b>	Se utiliza para Spring Security.
<b>spring-boot-starter-mustache</b>	Se utiliza para crear aplicaciones web MVC utilizando vistas Moustache.
<b>Spring-boot-starter-data-jpa</b>	Se utiliza para Spring Data JPA con Hibernate.
<b>spring-boot-starter</b>	Se utiliza para el inicio del núcleo, incluido el soporte de configuración automática, el registro y YAML.
<b>Spring-boot-starter-groovy-templates</b>	Se utiliza para crear aplicaciones web MVC utilizando vistas de plantilla Groovy.
<b>spring-boot-starter-freemarker</b>	Se utiliza para crear aplicaciones web MVC utilizando vistas FreeMarker.
<b>spring-boot-starter-batch</b>	Se utiliza para Spring Batch.





<b>Spring-boot-starter-social-linkedin</b>	Se utiliza para Spring Social LinkedIn.
<b>spring-boot-starter-cache</b>	Se utiliza para el soporte de almacenamiento en caché de Spring Framework.
<b>Spring-boot-starter-data-solr</b>	Se utiliza para la plataforma de búsqueda Apache Solr con Spring Data Solr.
<b>Spring-boot-starter-data-mongodb</b>	Se utiliza para la base de datos orientada a documentos MongoDB y Spring Data MongoDB.
<b>spring-boot-starter-jooq</b>	Se utiliza para que jOOQ acceda a bases de datos SQL. Una alternativa a spring-boot-starter-data-jpa o spring-boot-starter-jdbc.
<b>spring-boot-starter-jta-narayana</b>	Se utiliza para Spring Boot Narayana JTA Starter.
<b>spring-boot-starter-cloud-connectors</b>	Se utiliza para Spring Cloud Connectors que simplifica la conexión a servicios en plataformas en la nube como Cloud Foundry y Heroku.
<b>spring-boot-starter-jta-bitronix</b>	Se utiliza para transacciones JTA utilizando Bitronix.
<b>Spring-boot-starter-social-twitter</b>	Se utiliza para Spring Social Twitter.
<b>Spring-boot-starter-data-rest</b>	Se utiliza para exponer repositorios de Spring Data sobre REST usando Spring Data REST.

#### Iniciadores de producción Spring Boot

Nombre	Descripción
<b>spring-boot-starter-actuator</b>	Se utiliza para el actuator de Spring Boot que proporciona funciones listas para producción para ayudarlo a monitorear y administrar su aplicación.
<b>spring-boot-starter-remote-shell</b>	Se utiliza para el shell remoto CRaSH para monitorear y administrar su aplicación a través de SSH. En desuso desde 1.5.

#### Arrancadores técnicos Spring Boot

Nombre	Descripción
<b>spring-boot-starter-undertow</b>	Se utiliza para Undertow como contenedor de servlets integrado. Una alternativa a spring-boot-starter-tomcat.
<b>spring-boot-starter-jetty</b>	Se utiliza para Jetty como contenedor de servlets integrado. Una alternativa a spring-boot-starter-tomcat.
<b>spring-boot-starter-logging</b>	Se utiliza para iniciar sesión mediante Logback. Iniciador de registro predeterminado.
<b>spring-boot-starter-tomcat</b>	Se utiliza para Tomcat como contenedor de servlets integrado. Iniciador de contenedor de servlet predeterminado utilizado por spring-boot-starter-web.





### spring-boot-starter-log4j2

Se utiliza para Log4j2 para el registro. Una alternativa a spring-boot-starter-logging.

## Parent de Spring Boot Starter

### Parent de Spring Boot Starter

Spring-boot-starter-parent es un iniciador de proyectos. Proporciona configuraciones predeterminadas para nuestras aplicaciones. Es utilizado internamente por todas las dependencias. Todos los proyectos de Spring Boot usan spring-boot-starter-parent como padre en el archivo pom.xml.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.4.0.RELEASE</version>
</parent>
```

Parent Poms nos permite administrar las siguientes cosas para múltiples proyectos y módulos secundarios:

- **Configuración:** nos permite mantener la consistencia de la versión de Java y otras propiedades relacionadas.
- **Gestión de dependencias:** controla las versiones de las dependencias para evitar conflictos.
- Codificación fuente
- Versión de Java predeterminada
- Filtrado de recursos
- También controla la configuración predeterminada del complemento.

Spring-boot-starter-parent hereda la gestión de dependencias de spring-boot-dependencies. Solo necesitamos especificar el número de versión de Spring





Boot. Si hay un requisito del motor de arranque adicional, podemos omitir con seguridad el número de versión.

## Spring Boot Starter para parent interno

Spring Boot Starter Parent define spring-boot-dependencies como un pom padre. Hereda la gestión de dependencias de spring-boot-dependencies.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>1.6.0.RELEASE</version>
<relativePath>../../spring-boot-dependencies</relativePath>
</parent>
```

### Pom principal predeterminado

```
<properties>
<java.version>1.8</java.version>
<resource.delimiter>@</resource.delimiter>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
<maven.compiler.source>${java.version}</maven.compiler.source>
<maven.compiler.target>${java.version}</maven.compiler.target>
</properties>
```

La sección de properties define los valores predeterminados de la aplicación. La versión predeterminada de Java es 1.8. También podemos anular la versión de Java especificando una propiedad `<java.version> 1.8 </java.version>` en el proyecto pom. El pom principal también contiene algunas otras configuraciones relacionadas con la codificación y la fuente. El marco de Spring Boot usa estos valores predeterminados en caso de que no los haya definido en el archivo application.properties.





## Gestión de complementos

El **spring-boot-starter-parent** especifica la configuración por defecto para una gran cantidad de plugins incluidos maven-failsafe-plugin, maven-jar-plugin and maven-surefire-plugin.

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-failsafe-plugin</artifactId>
<executions>
<execution>
<goals>
<goal>integration-test</goal>
<goal>verify</goal>
</goals>
</execution>
</executions>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-jar-plugin</artifactId>
<configuration>
<archive>
<manifest>
<mainClass>${start-class}</mainClass> <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
</manifest>
</archive>
</configuration>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId>
<configuration>
<includes>
<include>**/*Tests.java</include>
<include>**/*Test.java</include>
</includes>
<excludes>
<exclude>**/Abstract*.java</exclude>
</excludes>
</configuration>
</plugin>
```

## Dependencias de Spring Boot

La dependencia **spring-boot-starter-parent** hereda de **spring-boot-dependencies**, también comparte todas estas características. Por lo tanto, Spring Boot administra la lista de dependencias como parte de la administración de dependencias.





```
● ● ●

<properties>
<activemq.version>5.13.4</activemq.version>
...
<ehcache.version>2.10.2.2.21</ehcache.version>
<ehcache3.version>3.1.1</ehcache3.version>
...
<h2.version>1.4.192</h2.version>
<hamcrest.version>1.3</hamcrest.version>
<hazelcast.version>3.6.4</hazelcast.version>
<hibernate.version>5.0.9.Final</hibernate.version>
<hibernate-validator.version>5.2.4.Final</hibernate-validator.version>
<hikaricp.version>2.4.7</hikaricp.version>
<hikaricp-java6.version>2.3.13</hikaricp-java6.version>
<hornetq.version>2.4.7.Final</hornetq.version>
<hsqldb.version>2.3.3</hsqldb.version>
<htmlunit.version>2.21</htmlunit.version>
<httpasyncclient.version>4.1.2</httpasyncclient.version>
<httpclient.version>4.5.2</httpclient.version>
<httpcore.version>4.4.5</httpcore.version>
<infinispan.version>8.2.2.Final</infinispan.version>
<jackson.version>2.8.1</jackson.version>
...
<jersey.version>2.23.1</jersey.version>
<jest.version>2.0.3</jest.version>
<jetty.version>9.3.11.v20160721</jetty.version>
<jetty-jsp.version>2.2.0.v201112011158</jetty-jsp.version>
<spring-security.version>4.1.1.RELEASE</spring-security.version>
<tomcat.version>8.5.4</tomcat.version>
<undertow.version>1.3.23.Final</undertow.version>
<velocity.version>1.7</velocity.version>
<velocity-tools.version>2.0</velocity-tools.version>
<webjars-hal-browser.version>9f96c74</webjars-hal-browser.version>
<webjars-locator.version>0.32</webjars-locator.version>
<wsdl4j.version>1.6.3</wsdl4j.version>
<xml-apis.version>1.4.01</xml-apis.version>
</properties>
<prerequisites>
<maven>3.2.1</maven>
</prerequisites>
```

## Spring Boot Starter sin parent

En algunos casos, no necesitamos heredar spring-boot-starter-parent en el archivo pom.xml. Para manejar tales casos de uso, Spring Boot brinda la flexibilidad de seguir usando la administración de dependencias sin heredar el spring-boot-starter-parent.





```
● ● ●

<dependencyManagement>
<dependencies>
<dependency>
<!-- Import dependency management from Spring Boot -->
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>2.1.1.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

## Spring Boot Starter Web

Hay dos características importantes de spring-boot-starter-web:

- Es compatible para el desarrollo web.
- Configuración automática

Si queremos desarrollar una aplicación web, necesitamos agregar la siguiente dependencia en el archivo pom.xml:

```
● ● ●

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<version>2.2.2.RELEASE</version>
</dependency>
```

Starter of Spring web utiliza Spring MVC, REST y Tomcat como servidor integrado predeterminado. La dependencia única spring-boot-starter-web atrae de forma transitiva todas las dependencias relacionadas con el desarrollo web. También reduce el recuento de dependencias de compilación. Spring-boot-starter-web depende transitivamente de lo siguiente:



- 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

De forma predeterminada, `spring-boot-starter-web` contiene la siguiente dependencia del servidor Tomcat:



Spring-boot-starter-web configura automáticamente las siguientes cosas que se requieren para el desarrollo web:

- Servlet de despachador
- Página de error
- Web JAR para gestionar las dependencias estáticas
- Contenedor de servlet integrado

## Servidor web incorporado Spring Boot

Cada aplicación Spring Boot incluye un servidor integrado. El servidor integrado está integrado como parte de la aplicación implementable. La ventaja del servidor integrado es que no necesitamos un servidor preinstalado en el entorno. Con Spring Boot, el servidor integrado predeterminado es **Tomcat**. Spring Boot también admite otros dos servidores integrados:





- **Servidor Jetty**
- **Servidor Undertow**

## Usando otro servidor web incorporado

Para las aplicaciones de pila de **servlets**, **spring-boot-starter-web** incluye **Tomcat** al incluir **spring-boot-starter-tomcat**, pero podemos usar **spring-boot-starter-jetty** o **spring-boot-starter-undertow** en su lugar.

Para aplicaciones de pila **reactiva**, **spring-boot-starter-webflux** incluye **Reactor Netty** al incluir **spring-boot-starter-reactor-netty**, pero podemos usar **spring-boot-starter-tomcat**, **spring-boot-starter-jetty** o **spring-boot-starter-undertow** en su lugar.

### **Servidor Jetty**

Spring Boot también admite un servidor integrado llamado **Jetty Server**. Es un servidor HTTP y un contenedor de servlets que tiene la capacidad de servir contenido estático y dinámico. Se utiliza cuando se requiere comunicación de máquina a máquina.

Si queremos agregar el servidor Jetty en la aplicación, necesitamos agregar la dependencia **spring-boot-starter-jetty** en nuestro archivo pom.xml.

**Recuerde:** mientras usa el servidor Jetty en la aplicación, asegúrese de que el servidor Tomcat predeterminado esté **excluido** de **spring-boot-starter-web**. Evita el conflicto entre servidores.



```
● ● ●

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<exclusions>
<exclusion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

También podemos personalizar el comportamiento del servidor Jetty utilizando el archivo **application.properties** .

### Servidor Undertow

Spring Boot proporciona otro servidor llamado **Undertow** . También es un servidor web integrado como Jetty. Está escrito en Java y administrado y patrocinado por JBoss. Las principales ventajas del servidor Undertow son:

- Soporta HTTP / 2
- Soporte de actualización HTTP
- Soporte Websocket
- Proporciona soporte para Servlet 4.0
- Flexible
- Integrable

**Recuerde:** mientras usa el servidor Undertow en la aplicación, asegúrese de que el servidor Tomcat predeterminado esté **excluido** de **spring-boot-starter-web**. Evita el conflicto entre servidores.





```


<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<exclusions>
<exclusion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-undertow</artifactId>
</dependency>

```

También podemos personalizar el comportamiento del servidor Undertow usando el archivo **application.properties**.

## spring-boot-starter-web frente a spring-boot-starter-tomcat

Spring-boot-starter-web contiene las dependencias web de Spring que incluyen spring-boot-starter-tomcat. Spring-boot-starter-web contiene lo siguiente:

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

Mientras que **spring-boot-starter-tomcat** contiene todo lo relacionado con el servidor Tomcat.

- core
- el
- logging
- websocket



El starter-tomcat tiene las siguientes dependencias:

```
● ● ●

<dependency>
<groupId>org.apache.tomcat.embed</groupId>
<artifactId>tomcat-embed-core</artifactId>
<version>8.5.23</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>org.apache.tomcat.embed</groupId>
<artifactId>tomcat-embed-el</artifactId>
<version>8.5.23</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>org.apache.tomcat.embed</groupId>
<artifactId>tomcat-embed-websocket</artifactId>
<version>8.5.23</version>
<scope>compile</scope>
</dependency>
```

También podemos usar **spring-mvc** sin usar el servidor Tomcat integrado. Si queremos hacerlo, necesitamos excluir el servidor Tomcat usando la etiqueta **<exclusion>** , como se muestra en el siguiente código.

```
● ● ●

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





## Spring Data JPA

Spring Data es un proyecto Spring Source de alto nivel. Su propósito es unificar y facilitar el acceso a los diferentes tipos de almacenes de persistencia, tanto sistemas de bases de datos relacionales como almacenes de datos NoSQL.

Cuando implementamos una nueva aplicación, debemos centrarnos en la lógica empresarial en lugar de la complejidad técnica y el código estándar. Es por eso que la especificación Java Persistent API (JPA) y Spring Data JPA son extremadamente populares.

Spring Data JPA agrega una capa en la parte superior de JPA. Significa que Spring Data JPA usa todas las características definidas por la especificación JPA, especialmente la entidad, las asignaciones de asociación y las capacidades de consulta de JPA. Spring Data JPA agrega sus propias características, como la implementación sin código del patrón de repositorio y la creación de consultas de base de datos a partir del nombre del método.

## Spring Data JPA

Spring Data JPA maneja la mayor parte de la complejidad del acceso a la base de datos basada en JDBC y ORM (Object Relational Mapping). Reduce el código repetitivo requerido por JPA. Hace que la implementación de su capa de persistencia sea más fácil y rápida.

Spring Data JPA tiene como objetivo mejorar la implementación de capas de acceso a datos reduciendo el esfuerzo a la cantidad necesaria.

### Características de Spring Data JPA

Hay **tres** características principales de Spring Data JPA que son las siguientes:

- **Repositorio sin código:** es el patrón relacionado con la persistencia más popular. Nos permite implementar nuestro código comercial en un nivel de abstracción superior.
- **Código repetitivo reducido:** proporciona la implementación predeterminada para cada método mediante sus interfaces de repositorio. Significa que ya no es necesario implementar operaciones de lectura y escritura.





- **Consultas generadas:** otra característica de Spring Data JPA es la **generación de consultas de base de datos** basadas en el nombre del método. Si la consulta no es demasiado compleja, necesitamos definir un método en nuestra interfaz de repositorio con el nombre que comienza con **findBy**. Después de definir el método, Spring analiza el nombre del método y crea una consulta para él. Por ejemplo:

```

● ● ●

public interface EmployeeRepository extends CrudRepository<Employee, Long>
{
    Employee findByName(String name);
}

```

En el ejemplo anterior, ampliamos **CrudRepository** que usa dos genéricos: **Employee** y **Long**. El empleado es la **entidad** que se va a administrar y **Long** es el tipo de datos de la clave principal

Spring genera internamente una **consulta JPQL** (Java Persistence Query Language) basada en el nombre del método. La consulta se deriva de la firma del método. Establece el valor del parámetro de vinculación, ejecuta la consulta y devuelve el resultado.

Hay algunas otras características que son las siguientes:

- Puede integrar código de repositorio personalizado.
- Es un repositorio poderoso y una abstracción de mapeo de objetos personalizado.
- Es compatible con la auditoría transparente.
- Implementa una clase base de dominio que proporciona propiedades básicas.
- Es compatible con varios módulos como Spring Data JPA, Spring Data MongoDB, Spring Data REST, Spring Data Cassandra, etc.



## Repositorio de datos de Spring

Spring Data JPA proporciona **tres** repositorios que son los siguientes:

- **CrudRepository:** Ofrece **creación, lectura, actualización y eliminación** estándar. Contiene métodos como **findOne ()**, **findAll ()**, **save ()**, **delete ()**, etc.
- **PagingAndSortingRepository:** extiende **CrudRepository** y agrega los métodos **findAll**. Nos permite **ordenar y recuperar** los datos de forma paginada.
- **JpaRepository:** Es un **repositorio específico de JPA** Se define en **Spring Data Jpa**. Extiende tanto el repositorio **CrudRepository** como **PagingAndSortingRepository**. Agrega los métodos específicos de JPA, como **flush ()** para desencadenar un flush en el contexto de persistencia.



## Spring Boot Starter Data JPA

Spring Boot proporciona la dependencia **spring-boot-starter-data-jpa** para conectar la aplicación Spring con la base de datos relacional de manera eficiente. Spring-boot-starter-data-jpa usa internamente la dependencia **spring-boot-jpa** (desde Spring Boot versión 1.5.3).





```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
<version>2.2.2.RELEASE</version>
</dependency>
```

Las bases de datos están diseñadas con tablas / relaciones. Los enfoques anteriores (JDBC) implicaban escribir consultas SQL. En JPA, almacenaremos los datos de los objetos en una tabla y viceversa. Sin embargo, JPA evolucionó como resultado de un proceso de pensamiento diferente.

Antes de JPA, ORM era el término más comúnmente utilizado para referirse a estos marcos. Es la razón por la que Hibernate se llama marco ORM.

JPA nos permite mapear clases de aplicaciones a tablas en la base de datos.

- **Entity Manager:** una vez que definimos el mapeo, maneja todas las interacciones con la base de datos.
- **JPQL (Java Persistence Query Language):** proporciona una forma de escribir consultas para ejecutar búsquedas en entidades. Es diferente de las consultas SQL. Las consultas JPQL ya comprenden el mapeo que se define entre entidades. Podemos agregar condiciones adicionales si es necesario.
- **API Criteria:** Define una API basada en Java para ejecutar búsquedas en la base de datos.

### Hibernar frente a JPA

Hibernate es la implementación de JPA. Es el marco ORM más popular, mientras que JPA es una API que define la especificación. Hibernate comprende el mapeo que agregamos entre objetos y tablas. Asegura que los datos se recuperen / almacenen de la base de datos según el mapeo. También proporciona funciones adicionales en la parte superior de JPA.





## Ejemplo de Spring Boot JPA

En este ejemplo, usaremos la dependencia `spring-boot-starter-data-jpa` para crear una conexión con la base de datos H2.

**Paso 1:** Abra el resorte Initializr <https://start.spring.io/>.

**Paso 2:** proporcione el nombre del **grupo**. Hemos proporcionado **com.codigodata**.

**Paso 3:** proporcione el Id. Del **artefacto**. Hemos proporcionado **spring-boot-jpa-ejemplo**.

**Paso 4:** agregue las dependencias: **Spring Web, Spring Data JPA y H2 Database**.

**Paso 5:** Haga clic en el botón **Generar**. Cuando hacemos clic en el botón Generar, envuelve el proyecto en un archivo **Jar** y lo descarga al sistema local.

The screenshot shows the Spring Initializr web application. In the 'Project' section, 'Maven Project' and 'Java' are selected. Under 'Spring Boot', '2.6.0 (SNAPSHOT)' is selected. In the 'Project Metadata' section, the 'Group' is set to 'com.codigodata', 'Artifact' to 'spring-boot-jpa-ejemplo', 'Name' to 'spring-boot-jpa-ejemplo', 'Description' to 'Demo project for Spring Boot', and 'Package name' to 'com.codigodata.spring-boot-jpa-ejemplo'. The 'Packaging' is set to 'Jar'. In the 'Dependencies' section, 'Spring Web' is selected under 'WEB', 'Spring Data JPA' under 'SQL', and 'H2 Database' under 'DB'. At the bottom, there are 'GENERATE' and 'SHARE...' buttons.

**Paso 6: extraiga** el archivo Jar y péguelo en el espacio de trabajo de STS.

**Paso 7: Importe** la carpeta del proyecto a STS.

Archivo -> Importar -> Proyectos existentes de Maven -> Examinar -> Seleccione la carpeta `spring-boot-jpa-example` -> Finalizar





La importación lleva algún tiempo.

**Paso 8:** Cree un paquete con el nombre **com.codigodata.controller** en la carpeta **src / main / java** .

**Paso 9:** Cree una clase de controlador con el nombre **ControllerDemo** en el paquete **com.codigodata.controller** .

### **ControllerDemo.java**

```
package com.codigodata.controller;

import org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class ControllerDemo{

    @RequestMapping("/")
    public String home(){
        return "home.jsp";
    }
}
```

**Paso 10:** Cree otro paquete con el nombre **com.codigodata.model** en la carpeta **src / main / java**.

**Paso 11:** Cree una clase con el nombre **Usuario** en el paquete **com.javatpoint.model**.

### **Usuario.java**

```
package com.codigodata.model;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name="userdata")
```





```
public class Usuario{
    @Id
    private int id;
    private String username;
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getUname(){
        return username;
    }
    public void setUname(String username){
        this.username = username;
    }
    @Override
    public String toString(){
        return "User [id=" + id + ", uname=" + username +
    "]";
    }
}
```

Ahora necesitamos configurar la base de datos H2.

**Paso 12:** Abra el archivo **application.properties** y configure lo siguiente: **puerto, habilite la consola H2, la fuente de datos y la URL.**

#### **application.properties**

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

**Paso 13:** Cree un archivo **SQL** en la carpeta **src / main / resources**.





Haga clic derecho en la carpeta src / main / resources -> Nuevo -> Archivo ->  
Proporcione el **nombre del archivo** -> Finalizar

Hemos proporcionado el nombre de archivo **data.sql** e insertamos los siguientes datos en él.

### **data.sql**

```
insert into userdata values(101,'Tom');
insert into userdata values(102,'Andrew');
insert into userdata values(103,'Tony');
insert into userdata values(104,'Bob');
insert into userdata values(105,'Sam');
```

**Paso 14:** Cree una carpeta con el nombre **webapp** en la carpeta **src** .

**Paso 15:** Cree un archivo JSP con el nombre que le devolvimos en **ControllerDemo** . En ControllerDemo.java, hemos vuelto **home.jsp** .

### **home.jsp**

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<form action="addUser">
ID :<br />
<input type="text" name="t1"><br />
User name :<br />
<input type="text" name="t2"><br />
<input type="submit" value="Add">
</form>
</body>
```





</html>

**Paso 16:** Ejecute el archivo **SpringBootJpaExampleApplication.java**. Podemos ver en la consola que nuestra aplicación se está ejecutando con éxito en el puerto **8085**.

```
Tomcat started on port(s): 8085 (http) with context path ''
Started SpringBootJpaExampleApplication in 42.692 seconds (JVM running for 49.673)
Initializing Spring DispatcherServlet 'dispatcherServlet'
Initializing Servlet 'dispatcherServlet'
Completed initialization in 80 ms
```

**Paso 17:** Abra el navegador e invoque la URL `http://localhost:8085/h2-console/`. Muestra la clase de controlador, la URL de JDBC que hemos configurado en el archivo **application.properties** y el nombre de usuario predeterminado sa.

The screenshot shows the H2 Console login interface. The URL in the address bar is `localhost:8085/h2-console/login.jsp?jsessionid=c7b386f0f759c95595c25e30a58a44e9`. The form fields are as follows:

- Saved Settings: Generic H2 (Embedded)
- Setting Name: Generic H2 (Embedded)
- Driver Class: org.h2.Driver
- JDBC URL: jdbc:h2:mem:javatpoint (highlighted with a red box)
- User Name: sa
- Password: (empty field)
- Buttons: Connect, Test Connection

También podemos probar la conexión haciendo clic en el botón **Probar conexión**. Si la conexión es exitosa, muestra un mensaje Prueba exitosa.

**Paso 18:** Haga clic en el botón **Conectar**. Muestra la estructura de la tabla `userdata` que hemos definido en **User.java**.





The screenshot shows a database browser interface with the following structure:

- Database: jdbc:h2:mem:javatpoint
- Schemas:
  - USERDATA** (highlighted with a red box):
    - ID
    - USERNAME
    - Indexes
  - INFORMATION\_SCHEMA
  - Users
- Version: H2 1.4.200 (2019-10-14)

**Paso 19:** Ejecuta la siguiente consulta para ver los datos que hemos insertado en el archivo **data.sql**.

1. SELECCIONAR \* DE USERDATA;

The screenshot shows a SQL query tool with the following interface:

- Buttons: Run, Run Selected, Auto complete, Clear, SQL statement.
- SQL statement input field: `SELECT * FROM USERDATA;`
- Result area:
 

ID	USERNAME
101	John
102	Andrew
103	Tony
104	Bob
105	Sam

(5 rows, 22 ms)
- Buttons: Edit

[Descargar proyecto](#)

## Starter Actuator

**Spring Boot Actuator** es un subproyecto del Spring Boot Framework. Incluye una serie de características adicionales que nos ayudan a monitorear y administrar la aplicación Spring Boot. Contiene los puntos finales del actuador (el lugar donde viven los recursos). Podemos usar extremos **HTTP** y **JMX** para administrar y monitorear la aplicación Spring Boot. Si queremos obtener





funciones listas para producción en una aplicación, debemos usar el **actuador Spring Boot**.

## Características del actuador de arranque

Hay **tres** características principales de Spring Boot Actuator:

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

**Endpoint:** Los endpoints del actuador nos permiten monitorear e interactuar con la aplicación. Spring Boot proporciona varios puntos finales integrados. También podemos crear nuestro propio punto final. Podemos habilitar y deshabilitar cada punto final individualmente. La mayoría de las aplicaciones eligen **HTTP**, donde el Id del punto final, junto con el prefijo de **/actuator**, se asigna a una URL.

Por ejemplo, el punto final **/health** proporciona la información de salud básica de una aplicación. El actuador, de forma predeterminada, lo asignó a **/actuator/health**.

**Metrics:** Spring Boot Actuator proporciona métricas dimensionales al integrarse con el **micrómetro**. El micrómetro está integrado en Spring Boot. Es la biblioteca de instrumentación que impulsa la entrega de métricas de aplicaciones de Spring. Proporciona interfaces **independientes del proveedor** para **temporizadores, medidores, contadores, resúmenes de distribución y temporizadores de tareas largas** con un modelo de datos dimensional.

**Audit:** Spring Boot proporciona un marco de auditoría flexible que publica eventos en un **AuditEventRepository**. Publica automáticamente los eventos de autenticación si Spring-Security está en ejecución.

## Habilitación del actuador de arranque

Podemos habilitar el actuador inyectando la dependencia **spring-boot-starter-actuator** en el archivo pom.xml.

1. **<dependency>**





2. <**groupIdgroupId**>
3. <**artifactIdartifactId**>
4. <**versionversion**>
5. </**dependency**>

## Puntos finales del actuador de arranque

Los puntos finales del actuador nos permiten monitorear e interactuar con nuestra aplicación Spring Boot. Spring Boot incluye varios puntos finales integrados y también podemos agregar puntos finales personalizados en la aplicación Spring Boot.

La siguiente tabla describe los puntos finales más utilizados.

Identificación	Uso	Defecto
<b>actuator</b>	Proporciona una <b>página de descubrimiento</b> basada en true hipertexto para los otros puntos finales. Requiere que Spring HATEOAS esté en el classpath.	
<b>auditevents</b>	Expone información de eventos de auditoría para la true aplicación actual.	
<b>autoconfig</b>	Se utiliza para mostrar un informe de configuración true automática que muestra todos los candidatos a la configuración automática y la razón por la que "se aplicaron" o "no se aplicaron".	
<b>beans</b>	Se utiliza para mostrar una lista completa de todos los true Spring beans en su aplicación.	
<b>configprops</b>	Se utiliza para mostrar una lista recopilada de todas las true @ConfigurationProperties.	
<b>dump</b>	Se utiliza para realizar un volcado de subprocesos.	true
<b>env</b>	Se utiliza para exponer propiedades del true ConfigurableEnvironment de Spring.	
<b>flyway</b>	Se utiliza para mostrar las migraciones de la base de datos true de Flyway que se hayan aplicado.	
<b>health</b>	Se utiliza para mostrar información sobre el estado de la False aplicación.	
<b>info</b>	Se utiliza para mostrar información arbitraria de la False aplicación.	
<b>loggers</b>	Se utiliza para mostrar y modificar la configuración de los true registradores en la aplicación.	
<b>liquibase</b>	Se utiliza para mostrar las migraciones de la base de datos true Liquibase que se han aplicado.	





<b>metrics</b>	Se utiliza para mostrar información de métricas para la aplicación actual.	true
<b>mappings</b>	Se utiliza para mostrar una lista recopilada de todas las rutas de @RequestMapping.	true
<b>shutdown</b>	Se utiliza para permitir que la aplicación se cierre correctamente.	true
<b>trace</b>	Se utiliza para mostrar información de seguimiento.	true

Para Spring MVC, se utilizan los siguientes puntos finales adicionales.

Identificación	Descripción	Defecto
<b>docs</b>	Se utiliza para mostrar documentación, incluidas solicitudes y respuestas de ejemplo para los puntos finales del actuador.	Falso
<b>heapdump</b>	Se utiliza para devolver un archivo de volcado de pila hprof comprimido con GZip.	Cierto
<b>jolokia</b>	Se utiliza para exponer beans JMX a través de HTTP (cuando Jolokia está en la ruta de clase).	Cierto
<b>LogFile</b>	Se utiliza para devolver el contenido del archivo de registro.	Cierto
<b>Prometheus</b>	Se utiliza para exponer métricas en un formato que un servidor prometheus puede extraer. Requiere una dependencia de micrómetro-registro-prometheus.	Cierto

## Propiedades del actuador Spring Boot

Spring Boot habilita la seguridad para todos los puntos finales del actuador. Utiliza autenticación **basada en** formularios que proporciona **una identificación de usuario** como el usuario y una **contraseña** generada aleatoriamente. También podemos acceder a los puntos finales restringidos por el actuador personalizando la seguridad básica en los puntos finales. Necesitamos anular esta configuración mediante el **management.security.roles** property . Por ejemplo:

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





## Ejemplo de actuador de arranque de resorte

Entendamos el concepto de actuador a través de un ejemplo.

**Paso 1:** Abra Spring Initializr <https://start.spring.io/> y cree un proyecto **Maven**.

**Paso 2:** proporcione el nombre del **grupo**. Hemos proporcionado **com.javatpoint**.

**Paso 3:** proporcione el Id. Del **artefacto**. Hemos proporcionado el **ejemplo de actuador de arranque de resorte**.

**Paso 4:** agregue las siguientes dependencias: **Spring Web, Spring Boot Starter Actuator y Spring Data Rest HAL Browser**.

**Paso 5:** Haga clic en el botón **Generar**. Cuando hacemos clic en el botón Generar, envuelve todas las especificaciones relacionadas con el proyecto en un archivo **Jar** y lo descarga a nuestro sistema local.

The screenshot shows the Spring Initializr interface. The configuration is as follows:

- Project:** Maven Project
- Language:** Java
- Spring Boot:** 2.2.2
- Project Metadata:** Group: com.javatpoint, Artifact: spring-boot-actuator-example
- Dependencies:** Spring Web (selected), Spring Boot Actuator (selected)

At the bottom, there is a red box around the "Generate - Ctrl + F" button.

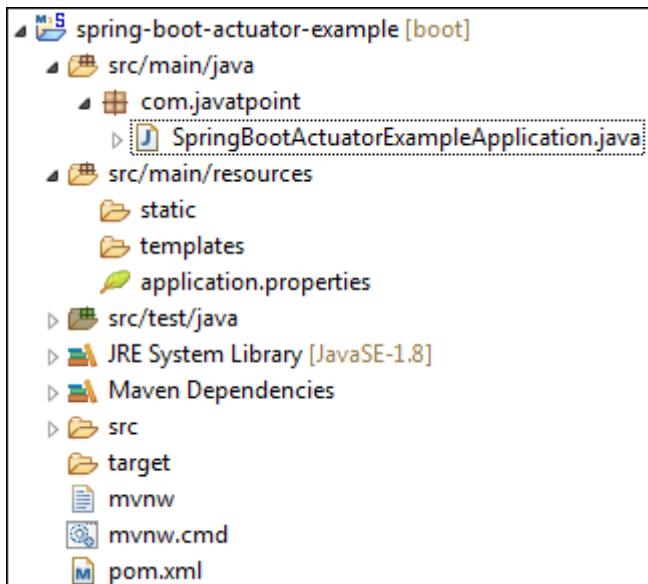
**Paso 6:** extraiga el archivo Jar y péguelo en el espacio de trabajo de STS.



**Paso 7:** Importa la carpeta del proyecto.

Archivo -> Importar -> Proyectos Maven existentes -> Examinar -> Seleccione la carpeta spring-boot-actuator-example -> Finalizar

La importación lleva algún tiempo. Después de importar el proyecto, podemos ver el directorio del proyecto en la sección del explorador de paquetes.



**Paso 8:** Cree una clase de controlador. Hemos creado la clase de controlador con el nombre DemoRestController.

#### DemoRestController.java

```
package com.codigodata.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class DemoRestController{
    @GetMapping("/hello")
    public String hello() {
        return "Hello User!";
    }
}
```





}

}

**Paso 9:** Abra el archivo **application.properties** y desactive la función de seguridad del actuador agregando la siguiente declaración.

### application.properties

1. **management.security.enabled = false**

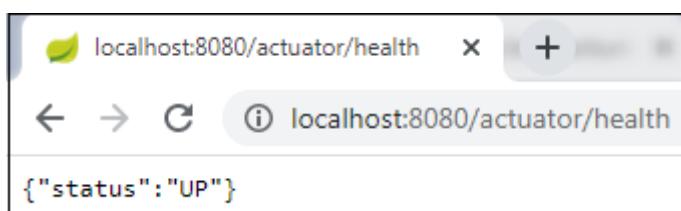
**Paso 10:** Ejecute el archivo **SpringBootActuatorExampleApplication.java**.

**Paso 11:** Abra el navegador e invoque la URL `http://localhost:8080/actuator`. Devuelve la siguiente página:

```
{"_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}}
```

La aplicación se ejecuta en el puerto 8080 de forma predeterminada. Una vez que se ha iniciado el actuador, podemos ver la lista de todos los puntos finales expuestos a través de HTTP.

Invoquemos el punto final de **health** invocando la URL `http://localhost:8080/actuator/health`. Denota el estado **up**. Significa que la aplicación está en buen estado y funcionando sin interrupciones.



De manera similar, podemos invocar otros puntos finales que nos ayuden a monitorear y administrar la aplicación Spring Boot.



[Descargar proyecto](#)

## Prueba de arranque de Spring Boot

El **motor de arranque-pruebas** es la dependencia principal de la prueba. Contiene la mayoría de los elementos necesarios para nuestras pruebas.

Hay varios tipos diferentes de pruebas que podemos escribir para ayudar a probar y automatizar el estado de una aplicación. Antes de comenzar cualquier prueba, necesitamos integrar el marco de prueba.

Con Spring Boot, necesitamos agregar **starter** a nuestro proyecto, para las pruebas solo necesitamos agregar la dependencia **spring-boot-starter-test**.

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>**

Extrae todas las dependencias relacionadas con la prueba. Después de agregarlo, podemos crear una prueba unitaria simple. Podemos crear el proyecto Spring Boot a través de IDE o generarlo usando Spring Initializr.

**Nota:** Si está agregando la dependencia de prueba manualmente, agréguela al final del archivo pom.xml.

En la dependencia anterior, una cosa a tener en cuenta es que incluye el alcance de la prueba **<scope> test </scope>**. Significa que cuando la aplicación está empaquetada y empaquetada para su implementación, se ignora cualquier dependencia que se declare con los ámbitos de prueba. Las dependencias del



alcance de la prueba solo están disponibles cuando se ejecutan en los modos de desarrollo y prueba de Maven.

Cuando creamos una aplicación Spring Boot simple, de forma predeterminada, contiene la dependencia de prueba en el archivo pom.xml y el archivo **ApplicationNameTest.java** en la carpeta **src / test / java**.

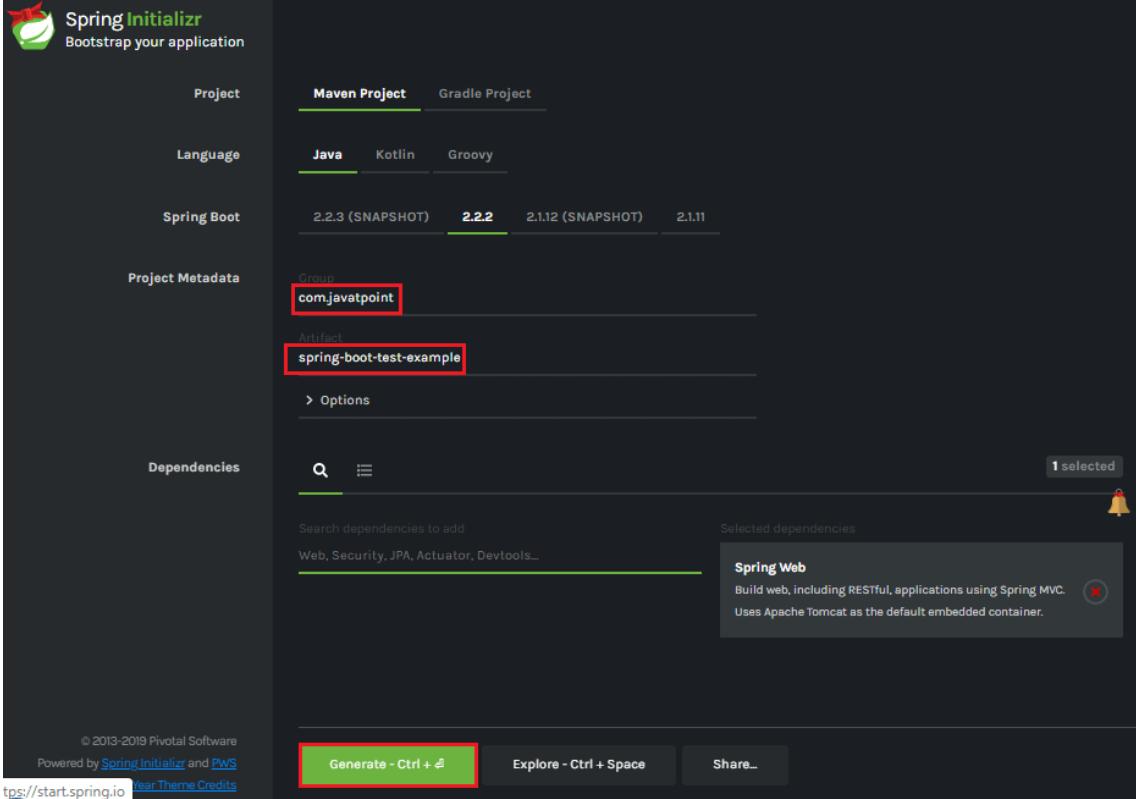
Creemos un proyecto maven simple.

## Ejemplo de prueba de Spring Boot Starter

**Paso 1:** Abra Spring Initializr <https://start.spring.io/>.

**Paso 2:** proporcione el nombre del **grupo** y la identificación del **artefacto**. Hemos proporcionado el nombre de grupo **com.javatpoint** y Artifact **spring-boot-test-example**.

**Paso 3:** agregue la dependencia de Spring Web.



The screenshot shows the Spring Initializr interface with the following configuration:

- Project:** Maven Project
- Language:** Java
- Spring Boot:** 2.2.2
- Project Metadata:**
  - Group: com.javatpoint
  - Artifact: spring-boot-test-example
- Dependencies:**
  - Search dependencies to add: Web, Security, JPA, Actuator, Devtools...
  - Selected dependencies: Spring Web (Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.)

At the bottom, there are buttons for "Generate - Ctrl + F" (highlighted in green), "Explore - Ctrl + Space", and "Share...".





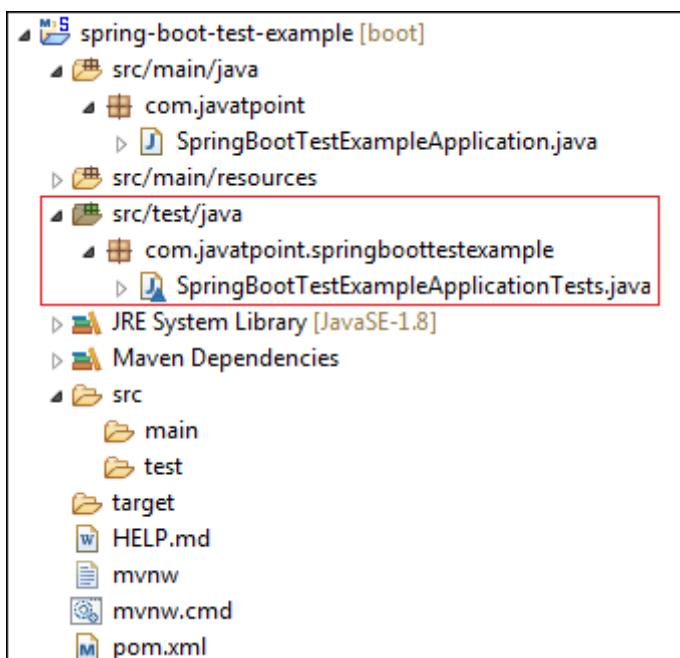
**Paso 4:** Haga clic en el botón **Generar**. Cuando hacemos clic en el botón Generar, envuelve todas las especificaciones relacionadas con el proyecto y descarga un archivo **Jar** a nuestro sistema local.

**Paso 5:** extraiga el archivo Jar descargado.

**Paso 6:** Importe la carpeta a STS. La importación lleva algún tiempo.

Archivo -> Importar -> Proyectos existentes de Maven -> Examinar -> Seleccione la carpeta spring-boot-test-example -> Finalizar

Después de importar el proyecto, podemos ver el siguiente directorio del proyecto en la sección Explorador de paquetes del STS.



Podemos ver en el directorio anterior que contiene un archivo de prueba llamado **SpringBootTestExampleApplicationTest.java** en la carpeta **src / test / java**.

### SpringBootTestExampleApplicationTest.java

1. **package** com.javatpoint.springbootexample;
2. **import** org.junit.jupiter.api.Test;
3. **import** org.springframework.boot.test.context.SpringBootTest;
4. **@SpringBootTest**



```

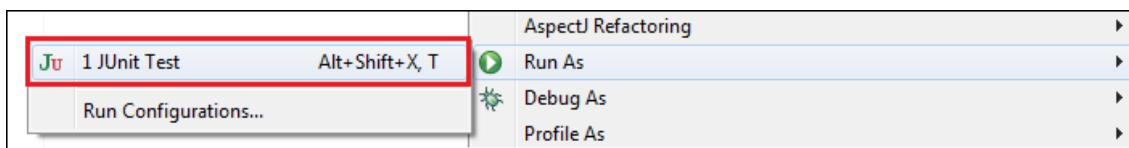
5. class SpringBootExampleApplicationTests
6. {
7.     @Test
8.     void contextLoads()
9.     {
10.    }
11.}

```

El código anterior implementa **dos** anotaciones de forma predeterminada: **@SpringBootTest** y **@Test**.

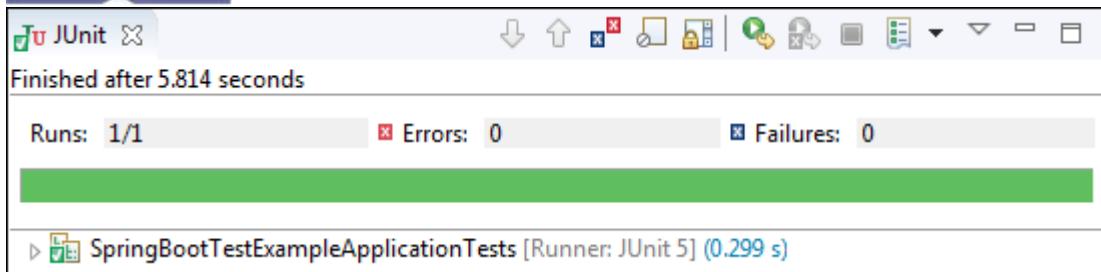
- **@SpringBootTest:** se aplica a una clase de prueba que ejecuta pruebas basadas en Spring Boot. Proporciona las siguientes características además del marco Spring TestContext Framework:
  - Utiliza **SpringBootTestLoader** como ContextLoader predeterminado si no se define una @ContextConfiguration (loader = ...) específica.
  - Busca automáticamente una **@SpringBootConfiguration** cuando no se usa la @Configuartion anidada y no se especifican clases explícitas.
  - Proporciona soporte para diferentes modos de **entorno web** .
  - Registra un **bean TestRestTemplate** o WebTestClient para su uso en pruebas web que utilizan el servidor web.
  - Permite definir los argumentos de la aplicación mediante el **atributo args**.

**Paso 7:** Abra el archivo **SpringBootExampleApplicationTest.java** y ejecútelo como **Junit Test**.



Cuando ejecutamos el código anterior, muestra lo siguiente:





[Descargar proyecto](#)

## Spring Boot DevTools

## Spring Boot DevTools

Spring Boot 1.3 proporciona otro módulo llamado Spring Boot DevTools. DevTools son las siglas de **Developer Tool**. El objetivo del módulo es intentar mejorar el tiempo de desarrollo mientras se trabaja con la aplicación Spring Boot. Spring Boot DevTools recoge los cambios y reinicia la aplicación.

Podemos implementar DevTools en nuestro proyecto agregando la siguiente dependencia en el archivo pom.xml.

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

## Funciones de Spring Boot DevTools

Spring Boot DevTools proporciona las siguientes características:

- **Valores predeterminados de propiedad**
- **Reinicio automático**
- **LiveReload**
- **Túnel de depuración remota**
- **Actualización y reinicio remotos**





**Valores predeterminados de propiedad:** Spring Boot proporciona la tecnología de plantillas **Thymeleaf** que contiene la propiedad **spring.thymeleaf.cache**. Deshabilita el caché y nos permite actualizar páginas sin necesidad de reiniciar la aplicación. Pero configurar estas propiedades durante el desarrollo siempre crea algunos problemas.

Cuando usamos el módulo `spring-boot-devtools`, no estamos obligados a establecer propiedades. Durante el desarrollo del almacenamiento en caché de Thymeleaf, Freemarker, Groovy Templates se desactivan automáticamente.

**Nota:** Si no queremos aplicar valores predeterminados de propiedad en una aplicación, podemos establecer configprop: `spring.devtools.add-properties []` en `false` en el archivo `application.properties`.

**Reinicio automático:** Significa volver a cargar las clases de Java y configurarlo en el lado del servidor. Después de los cambios del lado del servidor, se implementó dinámicamente, se reinicia el servidor y se carga el código modificado. Se utiliza principalmente en aplicaciones basadas en microservicios. Spring Boot usa **dos** tipos de ClassLoaders:

- Las clases que no cambian (terceros Jars) se cargan en el **base ClassLoader**.
- Las clases que estamos desarrollando activamente se cargan en el **restart ClassLoader**

Cuando se reinicia la aplicación, el ClassLoader reiniciado se desecha y se completa uno nuevo. Por lo tanto, el ClassLoader base siempre está disponible y poblado.

Podemos deshabilitar el reinicio automático de un servidor usando la propiedad **spring.devtools.restart.enabled** en `false`.

## Recordar:

- DevTools siempre monitorea los recursos de classpath.
- Solo hay una forma de activar un reinicio es actualizar la ruta de clases.
- DevTools requería un cargador de clases de aplicación independiente para funcionar correctamente. De forma predeterminada, Maven bifurca el proceso de solicitud.
- El reinicio automático funciona bien con **LiveReload**.





- DevTools depende del gancho de cierre del contexto de la aplicación para cerrarlo durante el reinicio.

**LiveReload:** el módulo Spring Boot DevTools incluye un servidor integrado llamado **LiveReload**. Permite que la aplicación active automáticamente una actualización del navegador cada vez que realizamos cambios en los recursos. También se conoce como **actualización automática**.

**Nota:** Podemos deshabilitar LiveReload estableciendo la propiedad **spring.devtools.livereload.enabled** en **false**.

Proporciona extensiones de navegador para Chrome, Firefox y Safari. De forma predeterminada, LiveReload está habilitado. LiveReload funciona en la siguiente ruta:

- / META-INF / maven
- / META-INF / recourse
- /resources
- /static
- /public
- /template

También podemos deshabilitar la recarga automática en el navegador excluyendo las rutas anteriores. Por ejemplo:

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

Podemos ver la otra ruta adicional usando la propiedad **spring.devtools.restart.additional-path**. Por ejemplo:

1. **spring.devtools.restart.additional-routes = / ruta-a-carpeta**

Si queremos excluir una ruta adicional y queremos mantener los valores predeterminados, use la propiedad **spring.devtools.restart.additional-exclude**. Por ejemplo:

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





## Recordar

- Podemos ejecutar un servidor LiveReload a la vez.
- Antes de iniciar la aplicación, asegúrese de que no se esté ejecutando ningún otro servidor LiveReload.
- Si iniciamos varias aplicaciones desde IDE, solo es compatible con el primer LiveReload.

**Tunelización de depuración remota:** Spring Boot puede tunelizar JDWP (Java Debug Wire Protocol) a través de HTTP directamente a la aplicación. Incluso puede trabajar en la implementación de aplicaciones para proveedores de Internet Cloud que solo exponen los puertos 80 y 443.

**Actualización y reinicio remotos:** hay otro truco que ofrece DevTools: admite **actualizaciones y reinicios** remotos de aplicaciones . Supervisa la ruta de clases local para detectar cambios en los archivos y los envía a un servidor remoto, que luego se reinicia. También podemos usar esta función en combinación con LiveReload.

## Usando un archivo de activación

El reinicio automático a veces puede ralentizar el tiempo de desarrollo debido a reinicios frecuentes. Para eliminar este problema, podemos utilizar un **archivo de activación**. Spring Boot monitorea el archivo de activación y detecta modificaciones en ese archivo. Reinicia el servidor y vuelve a cargar todos los cambios anteriores.

Podemos implementar el archivo disparador en nuestra aplicación agregando la propiedad **spring.devtools.restart.trigger-file**. El archivo puede ser interno o externo. Por ejemplo:

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

## Ejemplo de Spring Boot DevTools

**Paso 1:** Cree un proyecto Maven usando Spring Initializr <https://start.spring.io/>.

**Paso 2:** proporcione el nombre del **grupo** y la identificación del **artefacto**. Hemos proporcionado el nombre de grupo **com.javatpoint** y el Id. De artefacto **spring-boot-devtools-example**.





**Paso 3:** agregue las siguientes dependencias: **spring-boot-starter-web** y **spring-boot-devtools**.

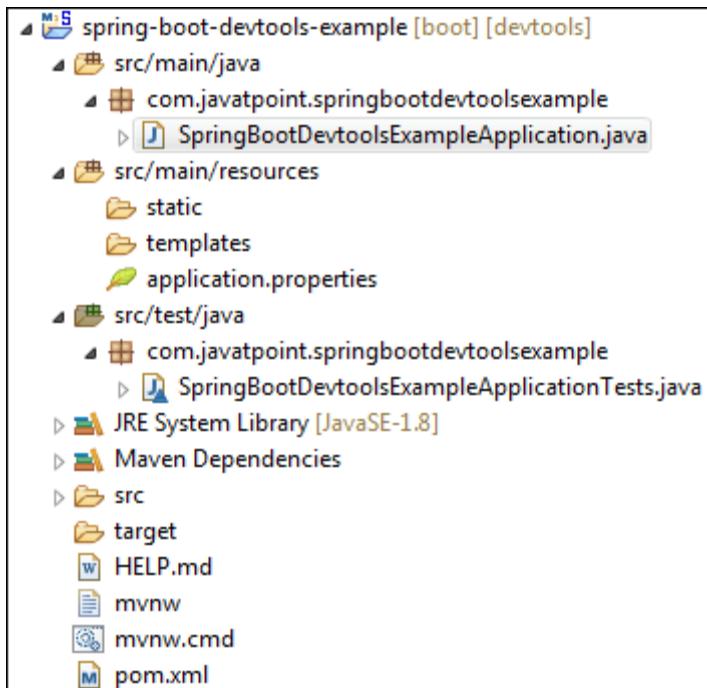
**Paso 4:** Haga clic en el botón **Generar**. Descarga el archivo **Jar** del proyecto.

**Paso 5:** extraiga el archivo Jar.

**Paso 6:** Importe la carpeta a STS. La importación lleva tiempo.

Archivo -> Importar -> Proyectos existentes de Maven -> Examinar -> Seleccione la carpeta `spring-boot-devtools-example` -> Finalizar

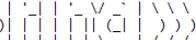
Cuando el proyecto se importa correctamente, podemos ver el siguiente directorio en la sección Explorador de paquetes del STS.



**Paso 7:** Abra **SpringBootDevtoolsExampleApplication.java** y ejecútelo como aplicación Java.

Después de eso, realice cualquier cambio (edite o elimine algún archivo o código) en la aplicación y guarde los cambios. Tan pronto como guardamos los cambios, el servidor se reinicia y recoge los cambios.



```
2019-12-27 12:46:40.154 INFO 8120 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2019-12-27 12:46:40.177 INFO 8120 --- [ restartedMain] o.s.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context root /
2019-12-27 12:46:40.178 INFO 8120 --- [ restartedMain] j.s.SpringBootDevtoolsExampleApplication : Started SpringBootDevtoolsExampleApplication
2019-12-27 12:46:40.181 INFO 8120 --- [ restartedMain] .ConditionEvaluationDeltaLoggingListener : Condition evaluation unchanged
2019-12-27 12:50:05.849 INFO 8120 --- [ Thread-34] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'applicationTaskExecutor'
  
  
Before changes  
  
  
After changes  
  
2019-12-27 12:50:06.688 INFO 8120 --- [ restartedMain] j.s.SpringBootDevtoolsExampleApplication : Starting SpringBootDevtoolsExampleApplication
2019-12-27 12:50:06.689 INFO 8120 --- [ restartedMain] j.s.SpringBootDevtoolsExampleApplication : No active profile set, falling back to default
2019-12-27 12:50:07.455 WARN 8120 --- [ restartedMain] org.apache.tomcat.util.modeler.Registry : The MBean registry cannot be disabled because it is being used by the application
2019-12-27 12:50:07.550 INFO 8120 --- [ restartedMain] o.s.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-12-27 12:50:07.557 INFO 8120 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-12-27 12:50:07.558 INFO 8120 --- [ restartedMain] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.24]
2019-12-27 12:50:07.622 INFO 8120 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-12-27 12:50:07.622 INFO 8120 --- [ restartedMain] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1 ms
2019-12-27 12:50:07.769 INFO 8120 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-12-27 12:50:07.851 INFO 8120 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2019-12-27 12:50:07.883 INFO 8120 --- [ restartedMain] o.s.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context root /
2019-12-27 12:50:07.885 INFO 8120 --- [ restartedMain] j.s.SpringBootDevtoolsExampleApplication : Started SpringBootDevtoolsExampleApplication
2019-12-27 12:50:07.887 INFO 8120 --- [ restartedMain] .ConditionEvaluationDeltaLoggingListener : Condition evaluation unchanged
```

## Descargar proyecto

# Proyecto Spring Boot Multi-Module

## Proyecto de varios módulos

Un proyecto de Spring Boot que contiene proyectos de maven anidados se denomina proyecto de **varios módulos**. En el proyecto de varios módulos, el proyecto principal funciona como un contenedor para las configuraciones de maven base.

En otras palabras, un **proyecto de varios módulos** se construye a partir de un pom principal que gestiona un grupo de submódulos. O Un **proyecto de varios módulos** se define mediante un POM principal que hace referencia a uno o más submódulos.

El proyecto principal de maven debe contener el tipo de empaquetado **pom** que hace que el proyecto sea un agregador. El archivo **pom.xml** del proyecto principal consiste en la lista de todos los **módulos, dependencias comunes y propiedades** que heredan los proyectos secundarios. El pom principal se encuentra en el directorio raíz del proyecto. Los módulos secundarios son proyectos Spring Boot reales que heredan las propiedades de maven del proyecto principal.





Cuando ejecutamos el proyecto de varios módulos, todos los módulos se implementan juntos en un servidor Tomcat integrado. También podemos implementar un módulo individual.

## POM padre

El POM principal define el **ID de grupo**, el **ID de artefacto**, la **versión** y los **paquetes java**. En los proyectos anteriores de Maven, hemos visto que el POM padre define los **jar** java del paquete específico. Pero en el proyecto de varios módulos, el **POM** principal define el paquete pom. El pom de paquetes se refiere a otros proyectos de Maven.

## Por qué necesitamos un proyecto de varios módulos

Dividir el proyecto en varios módulos es útil y fácil de mantener. También podemos editar o eliminar módulos fácilmente en el proyecto sin afectar a los otros módulos. Es útil cuando necesitamos implementar módulos individualmente.

Solo necesitamos especificar todas las dependencias en el pom padre. Todos los demás módulos comparten el mismo pom, por lo que no es necesario especificar la misma dependencia en cada módulo por separado. Hace que el código sea más fácil de mantener en orden con un gran proyecto.

## Módulo hijo module-ear, war, y jar

El módulo hijo puede ser cualquier proyecto y puede tener cualquier paquete. Somos libres de crear cualquier tipo de dependencia entre módulos y paquetes juntos.

Por ejemplo, estamos creando un archivo **EAR** (Enterprise ARchive), **WAR** (Web ARchive) y **JAR** (Java ARchive). Un archivo JAR se incluye en un archivo war que se incluye en un archivo EAR. El archivo EAR es el paquete final que está listo para implementarse en el servidor de aplicaciones.

El archivo EAR contiene uno o varios archivos WAR. Cada archivo WAR contiene el proyecto de servicio que tiene un código común para todos los archivos WAR y el tipo de empaquetado en el JAR.





.ear

.war

.jar

## Proyectos / módulos secundarios de Maven

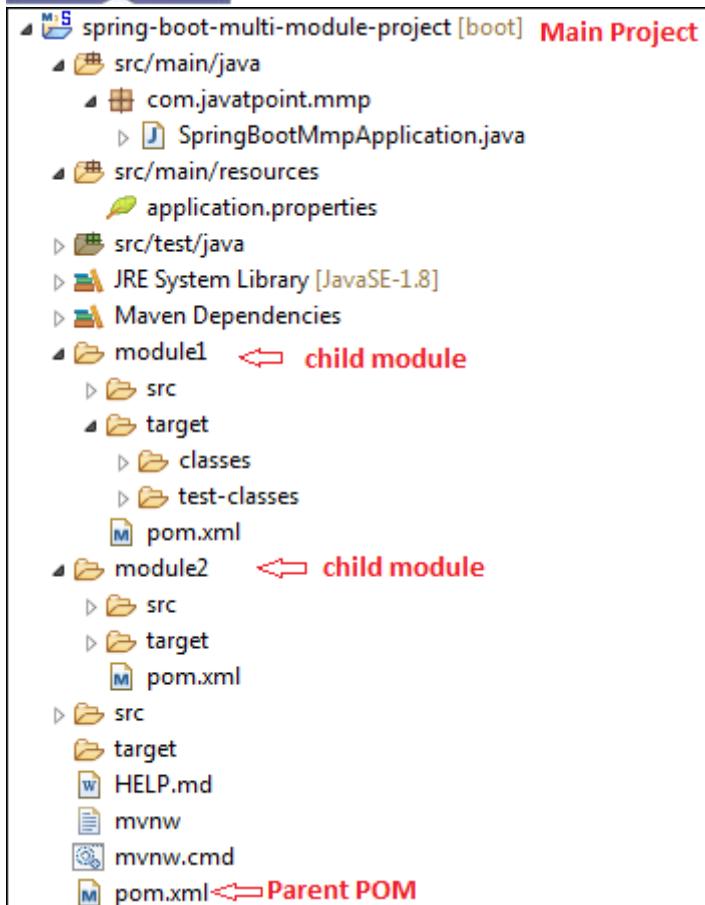
- Los módulos secundarios son proyectos independientes de maven que comparten propiedades del proyecto principal.
- Todos los proyectos secundarios se pueden construir con un solo comando porque está dentro de un proyecto principal.
- Es más fácil definir la relación entre los proyectos.

## Estructura de directorio de proyectos de varios módulos

Entendamos la estructura de directorios de proyectos de varios módulos.

En la siguiente imagen, hemos creado un proyecto llamado **spring-boot-multi-module-project**. Contiene el **pom** padre en la parte inferior del directorio. Después de eso, hemos creado dos **módulos Maven** llamados **module1** y **module2**, respectivamente. Estos dos módulos contienen sus propios archivos pom.





Abramos el POM principal y veamos qué configura cuando creamos módulos Maven en el proyecto.

### 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>

```

El archivo pom anterior es el mismo que hemos visto en los ejemplos anteriores. Pero en este archivo **pom**, se deben notar dos cosas: **packaging** y **modules**.

Cuando creamos un proyecto de varios módulos, necesitamos configurar el packaging pom en el archivo pom principal en lugar de jar.

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

Cuando creamos módulos Maven en el proyecto, Spring Boot configura automáticamente los módulos en el pom principal dentro de la etiqueta del **módulo**, como se muestra a continuación.

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





4. </modules>

Ahora, vamos a ver qué hay dentro del archivo pom de **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>



Aquí, un punto a tener en cuenta es que el archivo pom anterior no contiene las dependencias comunes como **starter-web**, **web-mvc**, etc. Hereda todas las dependencias comunes y las propiedades del **pom padre**.

## Ejemplo de proyecto de módulo múltiple Spring Boot

Creemos un ejemplo de una aplicación de varios módulos.

- **En el siguiente ejemplo, hemos creado un proyecto maven llamado spring-boot-multimodule. Es la aplicación principal. En la aplicación principal, hemos creado cinco módulos que son los siguientes:**
- **application**
- **model**
- **repository**
- **service-api**
- **service-impl**

### Módulo application

El módulo de aplicación es el módulo principal del proyecto. Contiene la clase de aplicación en la que se define el método principal que es necesario para ejecutar la aplicación Spring Boot. También contiene **propiedades de configuración de la aplicación, controlador, vistas y recursos**.

El módulo application incluye el módulo model, el módulo de implementación de servicio como dependencia que contiene el módulo de model, el módulo repository y el módulo de API de service.

### Módulo model

El módulo de modelo contiene **entidades y objetos visuales** que se utilizarán en el proyecto.

### Módulo de repository





El módulo Repository contiene **repositorios** que se utilizarán en el proyecto. Depende del módulo model.

### Módulo de API service

El módulo service-API contiene todos los **servicios del** proyecto . También depende del módulo de model.

### Módulo implementación services

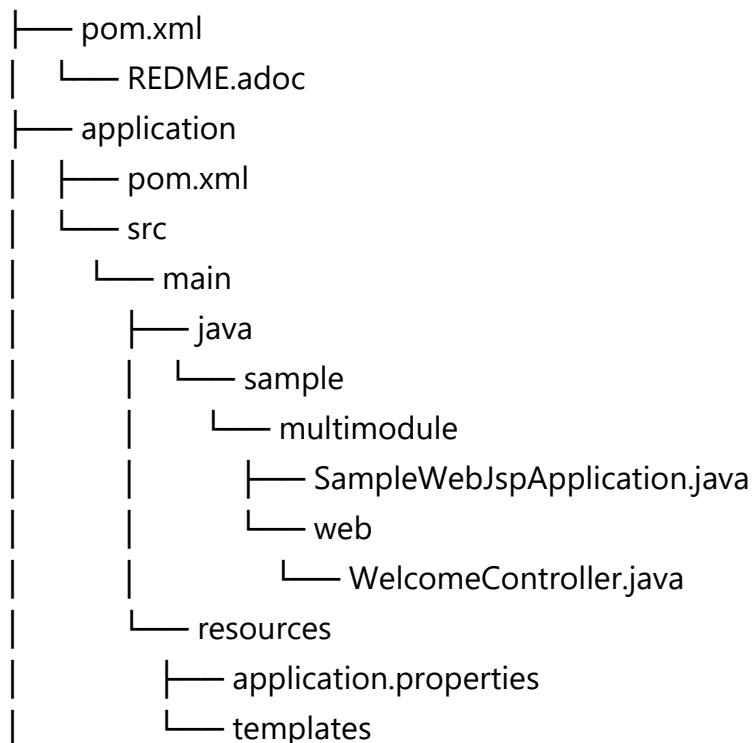
El módulo service-impl implementa el servicio. Depende del módulo repository y del módulo service-API.

## Agregador de POM (POM principal)

El pom principal contiene todos los módulos de la aplicación. También incluye todas las dependencias y propiedades comunes que necesita más de un módulo. Las dependencias se definen sin versión porque el proyecto ha definido Spring IO Platform como padre.

Entendamos la estructura de la aplicación de varios módulos que hemos creado.

### Spring-boot-multimodule





```
└── welcome
    └── show.html

└── model
    ├── pom.xml
    └── src
        └── main
            └── java
                └── sample
                    └── multimodule
                        └── domain
                            └── entity
                                └── Account.java

└── repository
    ├── pom.xml
    └── src
        └── main
            └── java
                └── sample
                    └── multimodule
                        └── repository
                            └── AccountRepository.java

└── service-api
    ├── pom.xml
    └── src
        └── main
            └── java
                └── sample
                    └── multimodule
                        └── service
                            └── api
                                ├── AccountNotFoundException.java
                                └── AccountService.java

└── service-impl
```





```

├── pom.xml
└── src
    └── main
        └── java
            └── sample
                └── multimodule
                    └── service
                        └── impl
                            └── AccountServiceImpl.java

```

**Paso 1:** Cree un **proyecto Maven** con el nombre **spring-boot-multimodule**.

**Paso 2:** Abra el archivo **pom.xml** (pom principal) y cambie el tipo de packaging **jar** a **pom**.

#### **pom.xml (pom principal)**

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>

```

Una cosa que debe notarse en el archivo pom anterior es que no hay ningún módulo maven configurado porque aún no lo hemos creado. Ahora crearemos los Módulos Maven uno por uno que hemos especificado anteriormente.

**Paso 3:** Cree un **módulo Maven** con la nombre **application** .

**Paso 4:** Abra el archivo **pom.xml** del módulo application y asegúrese de que el tipo de packaging sea **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>
19.
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>
34.
35.    </dependencies>
36.
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>
46.
47. </project>

```

**Paso 5:** crea la clase **principal** . Es la clase que se va a ejecutar.

### 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. }

```





**Paso 6:** Cree una clase de controlador con el nombre **WelcomeController** en el paquete **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.         // Tratando de obtener la cuenta
21.         Account account = accountService.findOne("23");
22.         if(account == null){
23.             // Si hay algún problema al crear la cuenta, regrese mostrar vista con estado de error
24.             model.put("message", "Error getting account!");
25.             model.put("account", "");
26.             return "welcome/show";
27.         }
28.         // Volver mostrar en la vista la información de cuenta como 23

```





```

29. String accountInfo = "Your account number is ".concat(account.getNumber());
   er());
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. }
```

**Paso 7:** cree un archivo **HTML** con el nombre **show.html** debajo de la carpeta **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>
```



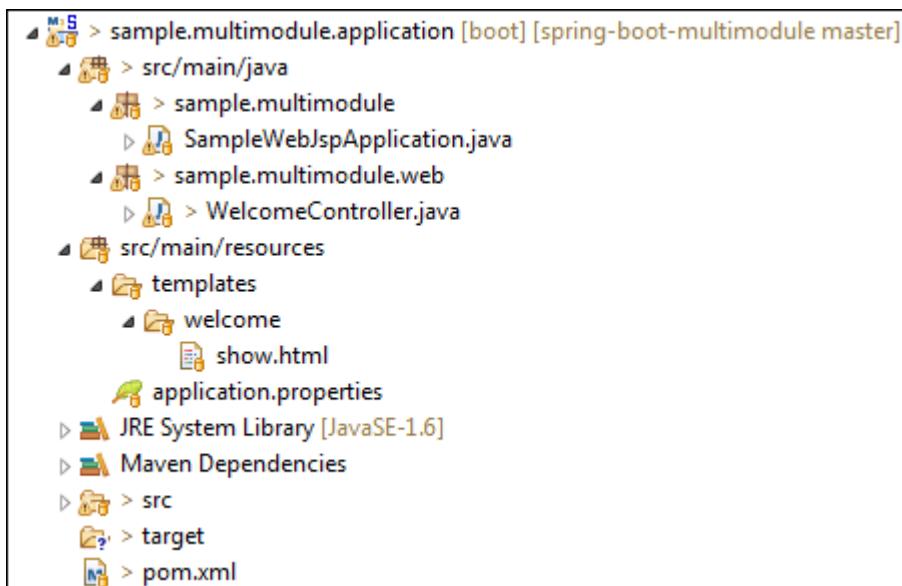


**Paso 8:** Abra el archivo **application.properties**, configure el **mensaje de la aplicación** y el caché de **thymeleaf** en **falso**.

### 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**

Después de crear todos los archivos anteriores, el directorio del módulo de la aplicación tiene el siguiente aspecto:



Creemos el segundo módulo que es **model**.

**Paso 9:** Cree un **módulo Maven** con el nombre **model** .

**Paso 10:** Abra el archivo **pom.xml** del módulo modelo y asegúrese de que el tipo de packaging sea **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> Módulo que contiene todas las Entidades y Objetos Visuales que se utilizarán en el proyecto. No tiene dependencias.`
13. `</description>`
14. `</project>`

**Paso 11:** Cree una clase con el nombre **Account** en el paquete **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;`





```
12.  
13.    private String number;  
14.  
15.    private String type;  
16.  
17.    private String creditCardNumber;  
18.  
19.    /**  
20.    * Crear un account vacío.  
21.    */  
22.    public Account() {  
23.  
24.    }  
25.  
26.    /**  
27.    * Crear un nuevo account.  
28.    *  
29.    * @param number  
30.    *      account tipo number  
31.    * @param id  
32.    *      account tipo id  
33.    */  
34.    public Account(Long id, String number) {  
35.        this.number = number;  
36.        this.id = id;  
37.    }  
38.  
39.    public Long getId() {  
40.        return id;  
41.    }  
42.  
43.    public void setId(Long id) {  
44.        this.id = id;  
45.    }
```

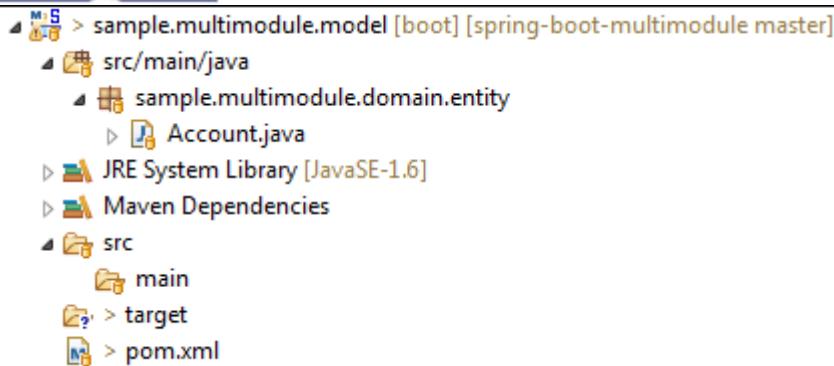




```
46.  
47. public String getNumber() {  
48.     return number;  
49. }  
50.  
51. public void setNumber(String number) {  
52.     this.number = number;  
53. }  
54.  
55. public String getType() {  
56.     return type;  
57. }  
58.  
59. public void setType(String type) {  
60.     this.type = type;  
61. }  
62.  
63. public String getCreditCardNumber() {  
64.     return creditCardNumber;  
65. }  
66.  
67. public void setCreditCardNumber(String creditCardNumber) {  
68.     this.creditCardNumber = creditCardNumber;  
69. }  
70.  
71.}
```

Después de crear todos los archivos anteriores, el directorio del módulo model tiene el siguiente aspecto:





Creamos el **tercer** módulo que es el **repository**.

**Paso 12:** Cree un **módulo Maven** con el nombre **repository** .

**Paso 13:** Abra el archivo **pom.xml** del módulo de la aplicación y asegúrese de que el tipo de packaging sea **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> Módulo que contiene todos los repositorios que se utilizarán en el proyecto. Depende del módulo model.</description>`
- 13.
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>
21.
22.    <!-- Spring Boot dependencies -->
23.    <dependency>
24.        <groupId>org.hsqldb</groupId>
25.        <artifactId>hsqldb</artifactId>
26.        <scope>runtime</scope>
27.    </dependency>
28.
29.    </dependencies>
30.
31. </project>

```

**Paso 14:** Cree una clase con el nombre **AccountRepository** en el paquete **sample.multimodule.repository**.

#### AccountRepository.java

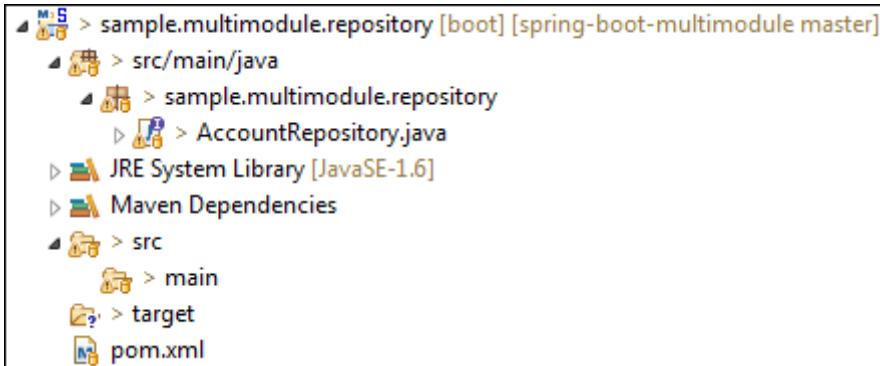
```

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, L
ong>
8. {
9.     Account findByNumber(String number);
10. }

```



Después de crear todos los archivos anteriores, el directorio del módulo repository tiene el siguiente aspecto:



Creemos el **cuarto** módulo que es **service-api**.

**Paso 15:** Cree un **módulo Maven** con el nombre **service-api**.

**Paso 16:** Abra el archivo **pom.xml** de la aplicación **service-api** y asegúrese de que el tipo de packaging sea **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> Módulo que contiene API de todos los servicios del
    proyecto. Depende del módulo model.</description>
13.
14. <dependencies>
15.
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>

```

**Paso 17:** Cree un paquete con el nombre **sample.multimodule.service.api**.

**Paso 18:** Cree una clase con el nombre **AccountNotFoundException**. Maneja la excepción si no se encuentra la cuenta.

#### **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 existe tal cuenta con id: " + accountId);
8.     }
9. }

```





**Paso 19:** Cree una clase con el nombre **AccountService**. Proporciona el servicio relacionado con la cuenta, como **buscar** y **crear** una cuenta.

### 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.      * Encuentra la cuenta con el número de cuenta proporcionado.
8.      *
9.      * @param number es el número de cuenta
10.     * @return retorna la cuenta
11.     * @throws AccountNotFoundException si no existe la cuenta.
12.    */
13.    Account findOne(String number) throws AccountNotFoundException;
14.    /**
15.     * Crea una nueva cuenta.
16.     * @param number
17.     * @return cuenta creada
18.    */
19.    Account createAccountByNumber(String number);
20. }

```

Después de crear todos los archivos anteriores, el directorio del módulo service-api tiene el siguiente aspecto:





```

M S > sample.multimodule.service.api [boot] [spring-boot-multimodule master]
  ▲ 📁 src/main/java
    ▲ 📁 sample.multimodule.service.api
      ▶ 📄 AccountNotFoundException.java
      ▶ 📄 AccountService.java
    ▶ 📄 JRE System Library [JavaSE-1.6]
    ▶ 📄 Maven Dependencies
  ▲ 📁 src
    📁 main
    📁 target
  pom.xml

```

**Paso 20:** Cree un **módulo Maven** con el nombre **service-impl**.

**Step 21:** Abra el archivo pom.xml de la aplicación service-impl y asegúrese de que el tipo de packaging sea jar. **pom.xml**

#### 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> Módulo que contiene la implementación de servicios definida en el módulo Service API. Depende del módulo repository y del módulo service-API.</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>

```

**Paso 22:** Cree una clase con el nombre **AccountServiceImpl** en el paquete **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. * Método dummy para fines de prueba
22. *
23. * @param number el número de cuenta. establece 0000 se obtiene
24. * un {@link AccountNotFoundException}
25. */
26. @Override
27. public Account findOne(String number) throws AccountNotFoundException {
28.     if(number.equals("0000")) {
29.         throw new AccountNotFoundException("0000");
30.     }
31.     Account account = accountRepository.findByNumber(number);
32.     if(account == null) {
33.         account = createAccountByNumber(number);
34.     }
35.     return account;
36. }
37. @Override
38. public Account createAccountByNumber(String number) {
39.     Account account = new Account();
40.     account.setNumber(number);
41.     return accountRepository.save(account);
42. }
43. public String getDummyType() {
44.     return dummyType;
45. }
46. public void setDummyType(String dummyType) {
47.     this.dummyType = dummyType;
48. }

```





Después de crear todos los archivos anteriores, el directorio del módulo **service-impl** tiene el siguiente aspecto:

```

M S > sample.multimodule.service.impl [boot] [spring-boot-multimodule master]
  ▲ 📂 src/main/java
    ▲ 📂 sample.multimodule.service.impl
      ▶ 📜 AccountServiceImpl.java
    ▶ 📂 JRE System Library [JavaSE-1.6]
    ▶ 📂 Maven Dependencies
  ▲ 📂 src
    📂 main
    📂 target
  pom.xml

```

Ahora abra el archivo **pom principal**, vemos que todos los módulos Maven que hemos creado están configurados en el pom principal dentro de la etiqueta. No es necesario que lo configuremos manualmente.

```

<modules>
  <module>model</module>
  <module>repository</module>
  <module>service-api</module>
  <module>service-impl</module>
  <module>application</module>
</modules>

```

Ahora asegúrese de que se hayan creado los cinco módulos, como se muestra a continuación:

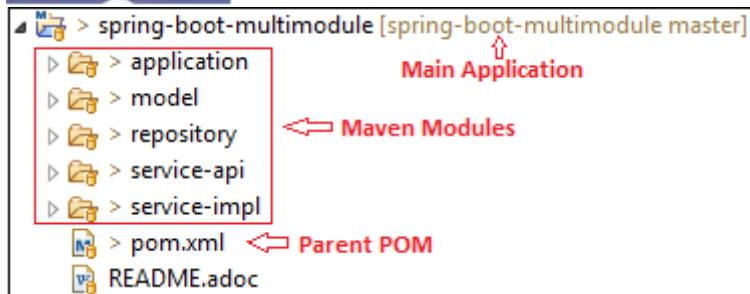
```

M S > sample.multimodule.application [boot] [spring-boot-multimodule master]
M S > sample.multimodule.model [boot] [spring-boot-multimodule master]
M S > sample.multimodule.repository [boot] [spring-boot-multimodule master]
M S > sample.multimodule.service.api [boot] [spring-boot-multimodule master]
M S > sample.multimodule.service.impl [boot] [spring-boot-multimodule master]

```

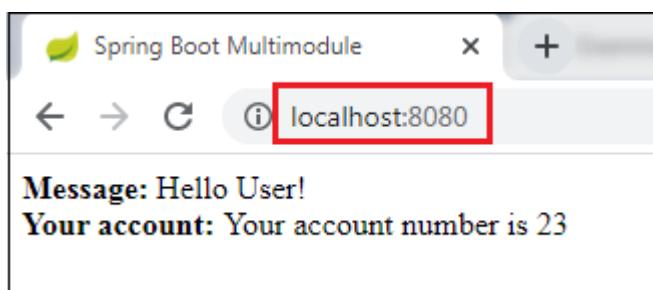
Después de crear todos los módulos, el directorio principal del proyecto tiene el siguiente aspecto:





**Paso 23:** Ahora ejecute el archivo **SampleWebJspApplication.java** como aplicación Java.

**Paso 24:** Abra el navegador e invoque la URL `http://localhost:8080`. Devuelve el **mensaje** y el número de cuenta que es **23**.

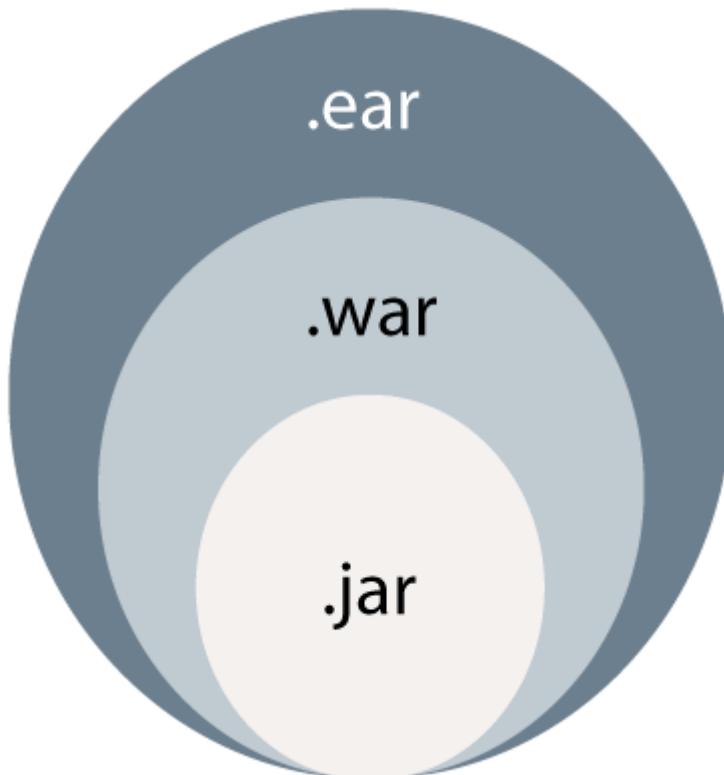


[Descargar proyecto](#)

## Spring Boot Packaging

En la aplicación J2EE, los módulos se empaquetan como **JAR**, **WAR** y **EAR**. Son los formatos de archivo comprimidos que se utilizan en J2EE. J2EE define tres tipos de archivos:

- WAR
- JAR
- EAR



## WAR

**WAR** son las siglas de **Web Archive**. El archivo WAR representa la aplicación web. El módulo web contiene clases de servlet, archivos JSP, archivos HTML, JavaScripts, etc. se empaquetan como un archivo JAR con extensión de **war**. Contiene un directorio especial llamado **WEB-INF**.

WAR es un módulo que se carga en un contenedor web del servidor de aplicaciones Java. El servidor de aplicaciones Java tiene **dos** contenedores: contenedor **web** y contenedor **EJB**.

El **contenedor web** aloja las aplicaciones web basadas en Servlet API y JSP. El contenedor web requiere que el módulo web esté empaquetado como un archivo WAR. Es un archivo JAR especial de archivo WAR que contiene un archivo **web.xml** en la carpeta **WEB-INF**.

Un **contenedor EJB** aloja beans Java empresariales basados en la API EJB. Requiere que los módulos EJB estén empaquetados como un archivo JAR. Contiene un archivo **ejb-jar.xml** en la carpeta **META-INF**.





La ventaja del archivo WAR es que se puede implementar fácilmente en la máquina cliente en un entorno de servidor web. Para ejecutar un archivo WAR, se requiere un servidor web o un contenedor web. Por ejemplo, Tomcat, Weblogic y Websphere.

## JAR

**JAR** son las siglas de **Java Archive**. Un módulo EJB (Enterprise Java Beans) que contiene archivos bean (archivos de clase), un manifiesto y un descriptor de implementación EJB (archivo XML) se empaquetan como archivos JAR con la extensión. **jar**. Los desarrolladores de software lo utilizan para distribuir clases de Java y varios metadatos.

En otras palabras, un archivo que encapsula una o más clases de Java, un manifiesto y un descriptor se denomina archivo JAR. Es el nivel más bajo del archivo. Se utiliza en J2EE para empaquetar EJB y aplicaciones Java del lado del cliente. Facilita la implementación.

## EAR

**EAR** son las siglas de **Enterprise Archive**. El archivo EAR representa la aplicación empresarial. Los dos archivos anteriores están empaquetados como un archivo JAR con la extensión. **ear**. Se implementa en el servidor de aplicaciones. Puede contener varios módulos EJB (JAR) y módulos web (WAR). Es un JAR especial que contiene un archivo **application.xml** en la carpeta **META-INF**.

# Spring Boot Auto-configuration

Auto-configuration configura automáticamente la aplicación Spring en función de las dependencias de jar que hemos agregado.

Por ejemplo, si el Jar de la base de datos H2 está presente en la ruta de clase y no hemos configurado ningún bean relacionado con la base de datos manualmente, la función de auto-configuration configura automáticamente en el proyecto.

Podemos habilitar la función de configuración automática usando la anotación **@EnableAutoConfiguration**. Pero esta anotación no se usa porque





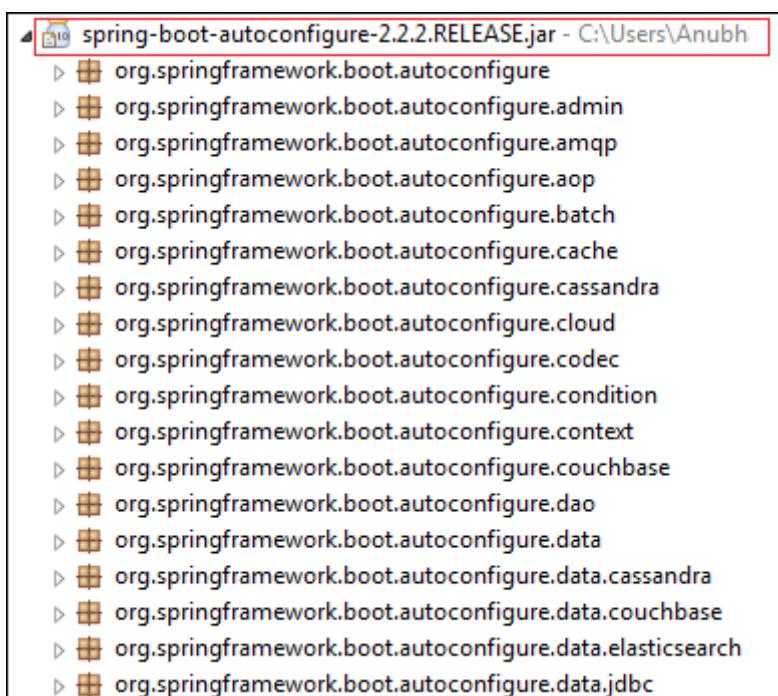
está envuelta dentro de la anotación **@SpringBootApplication**. La anotación **@SpringBootApplication** es la combinación de tres anotaciones: **@ComponentScan**, **@EnableAutoConfiguration** y **@Configuration**. Sin embargo, usamos la anotación **@SpringBootApplication** en lugar de usar **@EnableAutoConfiguration**.

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

Cuando agregamos la dependencia **spring-boot-starter-web** en el proyecto, la configuración automática de Spring Boot busca que Spring MVC esté en la ruta de clases. Configura automáticamente **dispatcherServlet**, una **página de error** predeterminada y **archivos web jars**.

De manera similar, cuando agregamos la dependencia **spring-boot-starter-data-jpa**, vemos que Spring Boot Auto-configuration, configura automáticamente una **fuente de datos** y un **Entity Manager**.

Toda la lógica de configuración automática se implementa en **spring-boot-autoconfigure.jar**, como se muestra en la siguiente figura.





## Necesidad de autoconfiguración

La aplicación basada en Spring requiere mucha configuración. Cuando usamos Spring MVC, necesitamos configurar **dispatcher servlet**, **view resolver**, **web jars**, entre otras cosas. El siguiente código muestra la configuración típica de un dispatcher servlet en una aplicación web:

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 >**

De manera similar, cuando usamos Hibernate / JPA, necesitamos configurar datasource, un transaction manager, una entity manager factory, entre muchas otras cosas.

### Configurando 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>`

### Configurando entity manager factory

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

### Configurando 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"/>`

## Deshabilitar clases de auto-configuration

También podemos deshabilitar las clases específicas de autoconfiguración, si no queremos que se apliquen. Usamos el atributo de **exclusión** de la anotación @EnableAutoConfiguration para deshabilitar las clases de configuración automática. Por ejemplo:

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. }
```

Podemos usar el atributo **excludeName** de la anotación `@EnableAutoConfiguration` y especificar el nombre **calificado** de la clase, si la clase no está en la ruta de clases. Podemos excluir cualquier cantidad de clases de configuración automática utilizando la propiedad `spring.autoconfigure.exclude` .

## Ejemplo de configuración automática de Spring Boot

En el siguiente ejemplo, veremos cómo funcionan las funciones de configuración automática de Spring Boot.

**Paso 1:** Abra el resorte Initializr <https://start.spring.io/> .

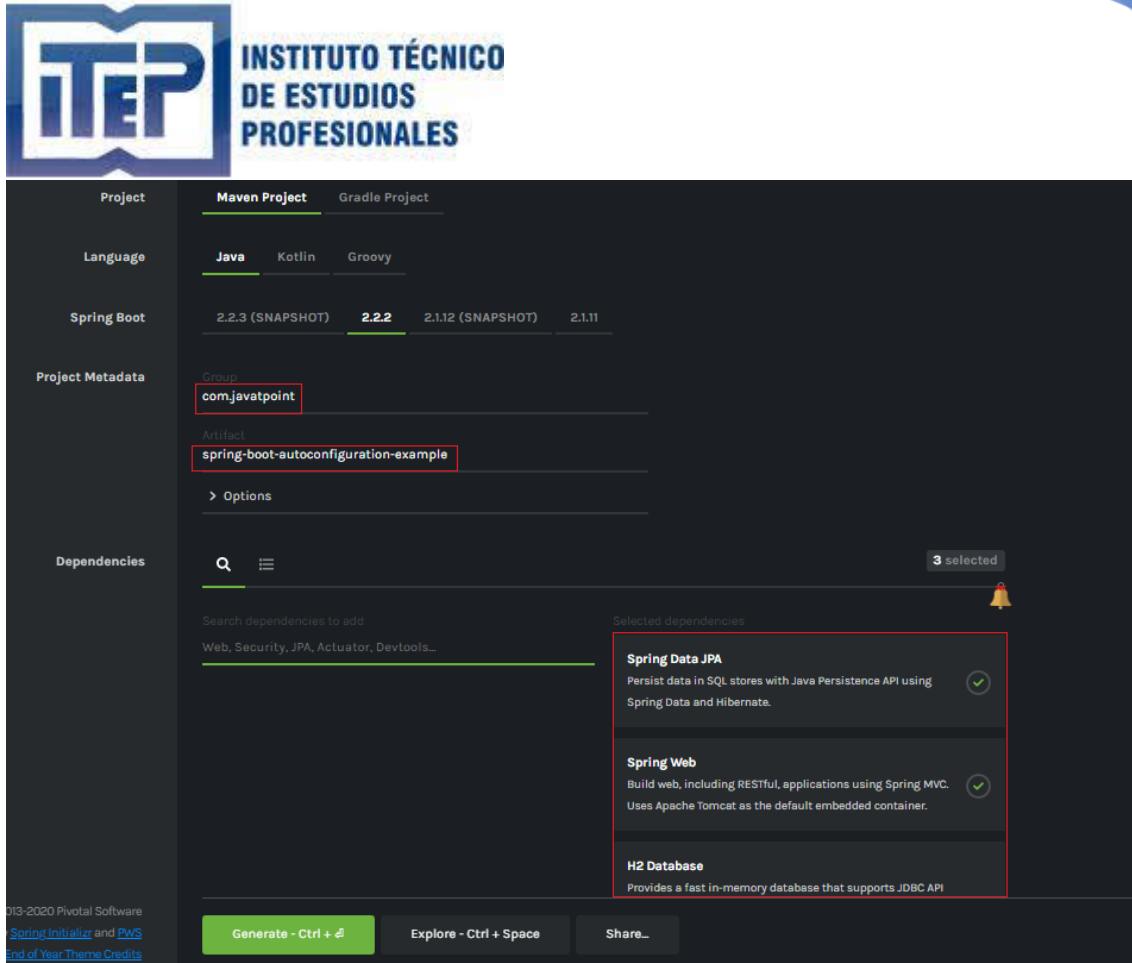
**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado **com.javatpoint** .

**Paso 3:** proporcione el Id. Del **artefacto** . Hemos proporcionado **spring-boot-autoconfiguration-example** .

**Paso 4:** agregue las dependencias: **Spring Web**, **Spring Data JPA**, an **H2 Database**.

**Paso 5:** Haga clic en el botón **Generar** . Cuando hacemos clic en el botón Generar, envuelve el proyecto en un archivo **Jar** y lo descarga al sistema local.





**Paso 6:** extraiga el archivo Jar y péguelo en el espacio de trabajo de STS.

**Paso 7: Importe** la carpeta del proyecto a STS.

Archivo -> Importar -> Proyectos Maven existentes -> Examinar -> Seleccione la carpeta **spring-boot-autoconfiguration-example** -> Finalizar

La importación lleva algún tiempo.

**Paso 8:** Cree un paquete con el nombre **com.javatpoint.controller** en la carpeta **src / main / java** .

**Paso 9:** Cree una clase de controlador con el nombre **ControllerDemo** en el paquete **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. }
```

**Paso 10:** Cree otro paquete con el nombre **com.javatpoint.model** en la carpeta **src / main / java** .

**Paso 11:** Cree una clase con el nombre **User** en el paquete **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 getUsername()
21. {
22.     return username;
23. }
24. public void setUsername(String username)
25. {
26.     this.username = username;
27. }
28. @Override
29. public String toString()
30. {
31.     return "User [id=" + id + ", uname=" + username + "]";
32. }
33. }
```

Ahora necesitamos configurar la base de datos H2.

**Paso 12:** Abra el archivo **application.properties** y configure lo siguiente: **puerto, habilite la consola H2, la fuente de datos y la URL.**

#### **application.properties**

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





**Paso 13:** Cree un archivo **SQL** en la carpeta **src / main / resources**.

Haga clic derecho en la carpeta **src / main / resources** -> Nuevo -> Archivo -> Proporcione el nombre del archivo -> Finalizar

Hemos proporcionado el nombre de archivo **data.sql** e insertamos los siguientes datos en él.

### **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');

**Paso 14:** Cree una carpeta con el nombre **webapp** en la carpeta **src** .

**Paso 15:** Cree un archivo JSP con el nombre que le devolvimos en **ControllerDemo** . En ControllerDemo.java, hemos vuelto **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>

**Paso 16:** Ejecute el archivo **SpringBootAutoconfigurationExampleApplication.java**. Podemos ver en la consola que nuestra aplicación se está ejecutando con éxito en el puerto **8085**.

```
Tomcat started on port(s): 8085 (http) with context path ''
Started SpringBootJpaExampleApplication in 42.692 seconds (JVM running for 49.673)
Initializing Spring DispatcherServlet 'dispatcherServlet'
Initializing Servlet 'dispatcherServlet'
Completed initialization in 80 ms
```

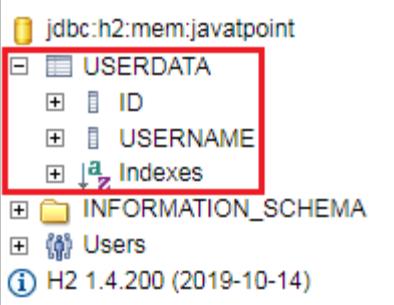
**Paso 17:** Abra el navegador e invoque la URL *http://localhost: 8085 / h2-console*/. Muestra la **clase de controlador**, la **URL de JDBC** que hemos configurado en el archivo **application.properties** y el nombre de usuario predeterminado **sa**.

The screenshot shows the H2 Console login interface. The URL in the address bar is `localhost:8085/h2-console/login.jsp?jsessionid=c7b386f0f759c95595c25e30a58a44e9`. The JDBC URL field contains the value `jdbc:h2:mem:javatpoint`, which is highlighted with a red box.

También podemos probar la conexión haciendo clic en el botón **Probar conexión**. Si la conexión es exitosa, muestra un mensaje **Prueba exitosa**.



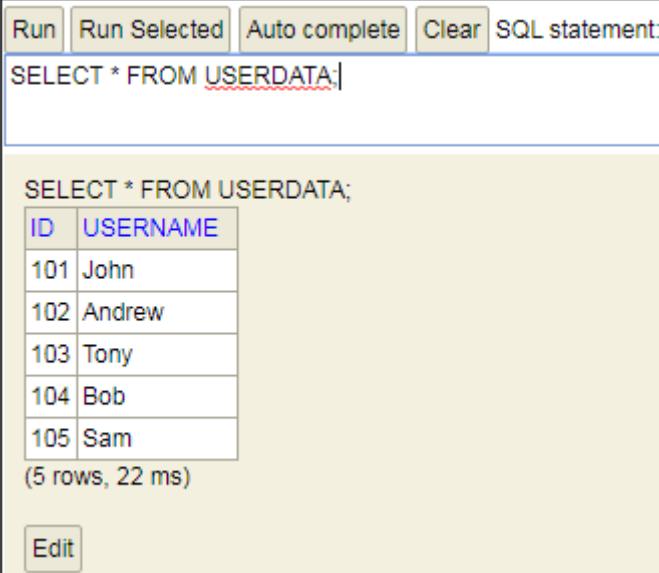
**Paso 18:** Haga clic en el botón **Conectar**. Muestra la estructura de la tabla **userdata** que hemos definido en el archivo User.java.



The screenshot shows the MySQL Workbench interface with the database browser open. A connection named "jdbc:h2:mem:javatpoint" is selected. Under the "USERDATA" table, the "Indexes" column is expanded, showing two entries: "a" and "z". Other tables like "INFORMATION\_SCHEMA" and "Users" are also listed, along with the H2 version information.

**Paso 19:** Ejecuta la siguiente consulta para ver los datos que hemos insertado en el archivo **data.sql**.

1. SELECCIONAR \* DE USERDATA;



The screenshot shows the MySQL Workbench SQL editor with the following content:

```
Run Run Selected Auto complete Clear SQL statement:
SELECT * FROM USERDATA;
```

Below the editor, the results of the query are displayed in a table:

ID	USERNAME
101	John
102	Andrew
103	Tony
104	Bob
105	Sam

(5 rows, 22 ms)

At the bottom left of the results area is an "Edit" button.

Echemos un vistazo de cerca a la consola. Vemos que **TransactionManagement**, **DispatcherServlet**, **EntityManagerFactory** y **DataSource** se configuran automáticamente, como se muestra en la siguiente figura.





```

gBootAutoconfigurationExampleApplication : Starting SpringBootAutoconfigurationExampleApplication on Anubhav-PC with PID 6192 (C:\Users\Anubhav\Documents\NetBeansProjects\gBootAutoconfigurationExample)
gBootAutoconfigurationExampleApplication : No active profile set, falling back to default profiles: default
.s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
.s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 30ms. Found 0 JPA repository interfaces.
trationDelegateBeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration' of type [null] registered in containing bean 'transactionManagementConfiguration' was not available during processing of bean 'org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration' at this point in time
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8085 (http)
o.apache.catalina.core.StandardService : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.29]
org.apache.jasper.servlet.TldScanner : At least one JAR was scanned for TLDs yet contained no TLDs. Enable debug logging for this logger for a complete list of scanned JARs.
o.a.c.c.C[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 3803 ms
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:javatpoint'
o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
org.hibernate.Version : HHH000412: Hibernate Core {5.4.9.Final}
o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformImpl]
j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during a flush.
o.s.s.concurrent.ThreadPoolExecutor : Tomcat started on port(s): 8085 (http) with context path ''
o.s.b.w.embedded.tomcat.TomcatWebServer : Started SpringBootAutoconfigurationExampleApplication in 8.359 seconds (JVM running for 9.92)
o.a.c.c.C[Tomcat].[localhost].[/] : [Initializing Spring DispatcherServlet 'dispatcherServlet']
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Completed initialization in 60 ms

```

## Depuración de la configuración automática

Podemos encontrar más información sobre la configuración automática utilizando las siguientes dos formas:

- Turning on debug logging
- Using Spring Boot Actuator

### Turning on debug logging

Podemos depurar el registro agregando una propiedad en el archivo **application.properties**. Implementemos el registro de depuración en el ejemplo anterior. Abra el archivo **application.properties** y agregue la siguiente propiedad:

1. `logging.level.org.springframework: DEBUG`

Ahora reinicia la aplicación. Vemos que se imprime un informe de configuración automática en el registro. El informe incluye todas las clases que se configuran automáticamente. Se divide en dos partes: **las coincidencias positivas** y **las coincidencias negativas**, como se muestra en la siguiente figura.

### Coincidencias positivas





```
=====
CONDITIONS EVALUATION REPORT
=====

Positive matches:
-----
AopAutoConfiguration matched:
- @ConditionalOnProperty (spring.aop.auto=true) matched (OnPropertyCondition)

AopAutoConfiguration.AspectJAutoProxyingConfiguration matched:
- @ConditionalOnClass found required class 'org.aspectj.weaver.Advice' (OnClassCondition)

AopAutoConfiguration.AspectJAutoProxyingConfiguration.CglibAutoProxyConfiguration matched:
- @ConditionalOnProperty (spring.aop.proxy-target-class=true) matched (OnPropertyCondition)

DataSourceAutoConfiguration matched:
- @ConditionalOnClass found required classes 'javax.sql.DataSource', 'org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType' (OnClassCondition)

DataSourceAutoConfiguration.PooledDataSourceConfiguration matched:
- AnyNestedCondition 1 matched 1 did not; NestedCondition on DataSourceAutoConfiguration.PooledDataSourceCondition.PooledDataSourceAvailable PooledDataSource
- @ConditionalOnBean (types: javax.sql.DataSource,Javax.sql.XADatasource; SearchStrategy: all) did not find any beans (OnBeanCondition)
```

## Coincidencias negativas

```
Negative matches:
-----
ActiveMQAutoConfiguration:
Did not match:
- @ConditionalOnClass did not find required class 'javax.jms.ConnectionFactory' (OnClassCondition)

AopAutoConfiguration.AspectJAutoProxyingConfiguration.JdkDynamicAutoProxyConfiguration:
Did not match:
- @ConditionalOnProperty (spring.aop.proxy-target-class=false) did not find property 'proxy-target-class' (OnPropertyCondition)

AopAutoConfiguration.ClassProxyingConfiguration:
Did not match:
- @ConditionalOnMissingClass found unwanted class 'org.aspectj.weaver.Advice' (OnClassCondition)
```

## Spring Boot Actuator

También podemos depurar la configuración automática utilizando **Actuator** en el proyecto. También agregaremos el **navegador HAL** para facilitar las cosas. Muestra los detalles de todos los beans que se configuran automáticamente y los que no.

Creemos un ejemplo de Spring Boot Actuator.

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado **com.javatpoint**.

**Paso 3:** proporcione el Id. Del **artefacto** . Hemos proporcionado **actuator-autoconfiguration-example**.

**Paso 4:** agregue las dependencias: **Spring Web** y **Spring Boot Actuator**.

1. <**dependency**>
2. <**groupIdgroupId**>
3. <**artifactIdartifactId**>



4. </dependency>
5. <dependency>
6. <groupId>org.springframework.boot</groupId>
7. <artifactId>spring-boot-starter-actuator</artifactId>
8. </dependency>

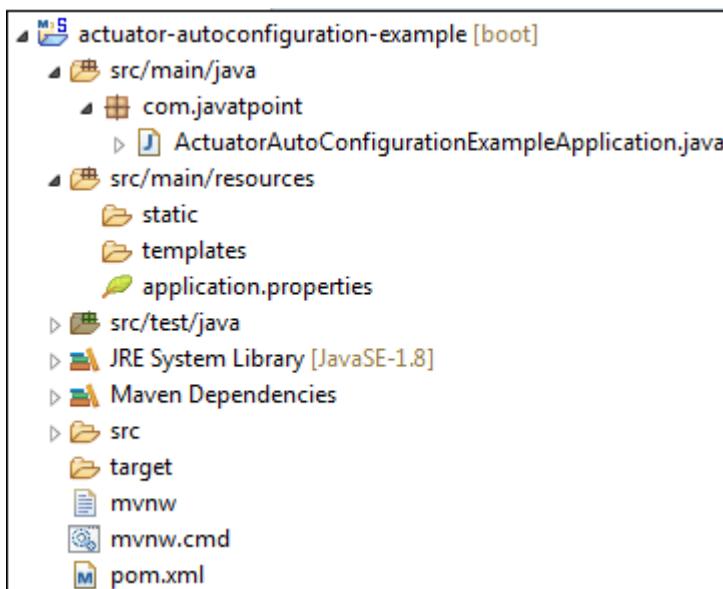
**Paso 5:** Haga clic en el botón **Generar**. Vincula todas las especificaciones relacionadas con el proyecto en un archivo **jar** y lo descarga en nuestro sistema local.

**Paso 6:** extraiga el archivo jar descargado.

**Paso 7: Importe** la carpeta del proyecto mediante los siguientes pasos:

Archivo -> Importar -> Proyecto Maven existente -> Siguiente -> Examinar -> Seleccione la carpeta del proyecto -> Finalizar

Después de importar el proyecto, podemos ver la siguiente estructura de directorios en la sección **Explorador de paquetes** del IDE.



**Paso 8:** Cree una clase de controlador en el paquete **com.javatpoint**. Hemos creado una clase de controlador con el nombre **DemoRestController**.



En el controlador, hemos definido un método llamado **hello ()** que devuelve una cadena.

### DemoRestController.java

```

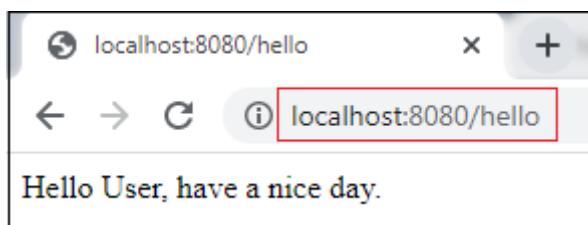
1. package com.javatpoint;
2. import org.springframework.web.bind.annotation.GetMapping;
3. import org.springframework.web.bind.annotation.RestController;
4. @RestController
5. public class DemoRestController
6. {
7.     @GetMapping("/hello")
8.     public String hello()
9.     {
10.        return "Hello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

#### Paso

**9:** Ejecute

el archivo **ActuatorAutoConfigurationExampleApplication.java** .

**Paso 10:** Abra el navegador e invoque la URL *http://localhost: 8080 / hello* . Devuelve una cadena que hemos especificado en el controlador.



Ahora invoque la URL del actuador *http://localhost: 8080 / actuator* . Lanza el actuador que muestra las tres URL: **auto** , **salud** e **información** , como se muestra a continuación.

```
{"_links": {"self": {"href": "http://localhost:8080/actuator", "templated": false}, "health": {"href": "http://localhost:8080/actuator/health", "templated": false}, "info": {"href": "http://localhost:8080/actuator/info", "templated": false}}
```





```
health-
path": {"href": "http://localhost:8080/actuator/health/{*path}", "templated": true}, "info": {"href": "http://localhost:8080/actuator/info", "templated": false}}
}
```

**Paso 11:** Abra el archivo **pom.xml** y agregue la dependencia del **navegador HAL**.

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

**Paso 12:** nuevamente, ejecute el archivo **ActuatorAutoConfigurationExampleApplication.java**.

Para acceder al navegador HAL, escriba *http://localhost: 8080* en el navegador y presione la tecla Intro.

The screenshot shows the HAL Browser interface. The top navigation bar includes tabs for 'The HAL Browser (for Spring Data REST)', 'Go To Entry Point', and 'About The HAL Browser (for Spring Data REST)'. The main area is divided into two main sections: 'Explorer' on the left and 'Inspector' on the right.

**Explorer:** This section contains several input fields and buttons:
 

- A search bar with placeholder '/'. Below it is a 'Go!' button.
- A 'Custom Request Headers' input field.
- A 'Properties' input field containing '{ }'.
- A 'Links' table with columns: rel, title, name / index, docs, GET, and NON-GET. It has one row labeled 'profile' with a green 'Ir' (Go) button.

**Inspector:** This section displays the 'Response Headers' and 'Response Body' for a successful 200 status code response.

- Response Headers:** Shows headers including date, transfer-encoding, and content-type.
- Response Body:** Shows a JSON object representing a link profile:
 

```
{
  "_links": {
    "profile": {
      "href": "http://localhost:8080/profile"
    }
  }
}
```

Ahora podemos acceder al actuador a través del navegador HAL.

Escriba **/ actuador** en el cuadro de texto del Explorador y haga clic en el botón **Ir**.



**INSTITUTO TÉCNICO  
DE ESTUDIOS  
PROFESIONALES**

The HAL Browser (for Spring Data) × +

localhost:8080/browser/index.html#/

The HAL Browser (for Spring Data REST) Go To Entry Point About The HAL

# Explorer



Muestra todas las cosas relacionadas con el actuador. Lo más importante en el actuador son los **frijoles**.

## Links

rel	title	name / index	docs	GET	NON-GET
self					
auditevents					
beans					
caches-cache					Follow link

Cuando hacemos clic en la flecha del bean, muestra todos los beans configurados en el proyecto Spring Boot.



The HAL Browser (for Spring Data REST)   Go To Entry Point   About The HAL Browser (for Spring Data REST)

```
{
  "contexts": {
    "application": {
      "beans": {
        "helloWorldController": {
          "aliases": [],
          "scope": "singleton",
          "type": "com.javatpoint.server.main.helloworld.HelloWorldController$$Enhance
rBySpringCGLIB$$d59d461d",
          "resource": "file [C:\\Users\\Anubhav\\Documents\\workspace-sts-3.9.9.RELEAS
E\\restful-web-services\\target\\classes\\com\\javatpoint\\server\\main\\helloworld\\H
elloWorldController.class]",
          "dependencies": ["messageSource"]
        },
        "pagedResourcesAssembler": {
          "aliases": [],
          "scope": "singleton",
          "type": "org.springframework.data.web.PagedResourcesAssembler",
          "resource": "class path resource [org/springframework/data/rest/webmvc/config/RepositoryRestMvcConfiguration.class]",
          "dependencies": ["pageableResolver"]
        },
        "serviceModelToSwagger2MapperImpl": {
          "aliases": [],
          "scope": "singleton",
          "type": "springfox.documentation.swagger2.mappers.ServiceModelToSwagger2Mapp
erImpl",
          "resource": "URL [jar:file:/C:/Users/Anubhav/.m2/repository/io/springfox/spr
infox-swagger2/2.9.2/springfox-swagger2-2.9.2.jar!/springfox/documentation/swagger2/m
appers/ServiceModelToSwagger2MapperImpl.class]",
          "dependencies": ["modelMapperImpl", "parameterMapperImpl", "securityMapperIm
pl", "licenseMapperImpl", "vendorExtensionsMapperImpl"]
        }
      }
    }
  }
}
```

La imagen de arriba muestra los detalles de todos los **beans** que están configurados automáticamente y los que no.

[Descargar proyecto de ejemplo de configuración automática](#)

[Descargar proyecto de ejemplo de configuración automática del actuador](#)

## Ejemplo de Spring Boot Hello World

En la sección, crearemos un proyecto de **Maven** para el ejemplo de Hello Word. Necesitamos las siguientes herramientas y tecnologías para desarrollar las mismas.

- Spring Boot 2.2.2 RELEASE
- JavaSE 1.8
- Maven 3.3.9
- STS IDE

**Paso 1:** Abra Spring Initializr <https://start.spring.io/> .

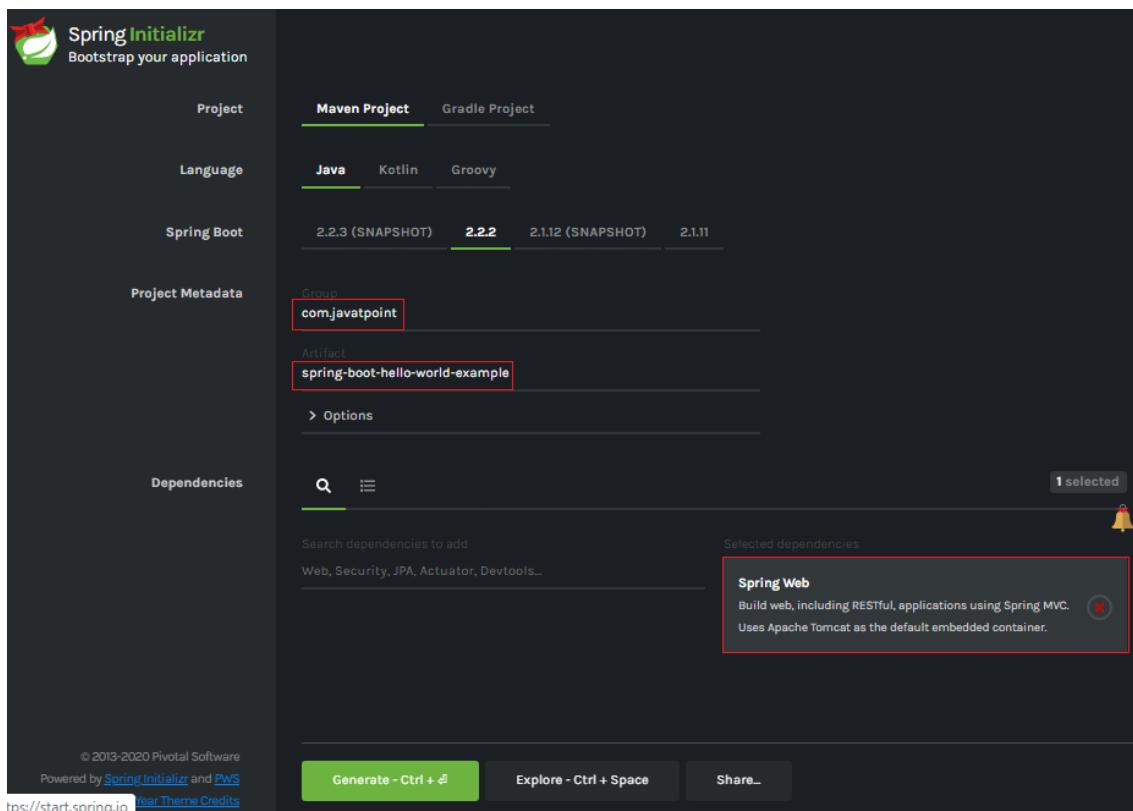
**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado **com.javatpoint**.



**Paso 3:** proporcione el Id. Del **artefacto** . Hemos proporcionado el **ejemplo de spring-boot-hello-world**.

**Paso 4:** agregue la dependencia **Spring Web**.

**Paso 5:** Haga clic en el botón **Generar** . Cuando hacemos clic en el botón Generar, envuelve todas las especificaciones en un archivo jar y lo descarga a nuestro sistema local.



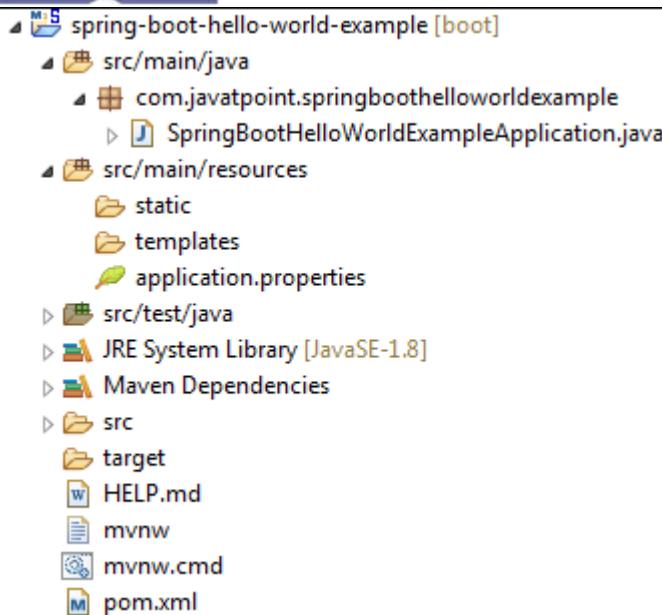
**Paso 6: extraiga el archivo RAR.**

**Paso 7: Importe** la carpeta del proyecto mediante los siguientes pasos:

Archivo -> Importar -> Proyecto Maven existente -> Siguiente -> Examinar -> Seleccione la carpeta del proyecto -> Finalizar

Cuando el proyecto se importa correctamente, muestra el siguiente directorio del proyecto en la sección Explorador de paquetes del IDE.





**Paso 8:** Cree un paquete con el nombre **com.javatpoint.controller** dentro de la carpeta **src / main / java**.

**Paso 9:** Cree una clase de controlador con el nombre **HelloWorldController**.

**Paso 10:** cree un método llamado **hello ()** que devuelva una cadena.

### 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. }





**Paso 11:** Ejecute el archivo **SpringBootHelloWorldExampleApplication.java**.

### SpringBootHelloWorldExampleApplication.java

```

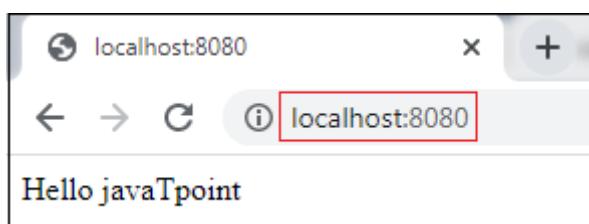
1. package com.javatpoint;
2. import org.springframework.boot.SpringApplication;
3. import org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. public class SpringBootHelloWorldExampleApplication
6. {
7.     public static void main(String[] args)
8.     {
9.    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

Cuando la aplicación se ejecuta correctamente, muestra un mensaje en la consola, como se muestra en la siguiente figura.

```

: Initializing ExecutorService 'applicationTaskExecutor'
: Tomcat started on port(s): 8080 (http) with context path ''
: Started SpringBootHelloWorldExampleApplication in 4.526 seconds |(JVM running for 5
: Initializing Spring DispatcherServlet 'dispatcherServlet'
: Initializing Servlet 'dispatcherServlet'
: Completed initialization in 44 ms
```

**Paso 12:** Abra el navegador e invoque la URL **https://localhost:8080**. Devuelve una cadena que hemos especificado en el controlador.



[Descargar proyecto de ejemplo de Hello World](#)



# Implementación de proyectos con Tomcat

En esta sección, aprenderemos cómo implementar la aplicación Spring Boot en Tomcat Server.

Incluye tres pasos:

- **Configuración de una aplicación Spring Boot**
- **Crear un WAR de Spring Boot**
- **Implementar WAR en Tomcat**

## Ejemplo

Creemos un ejemplo de Maven para implementarlo en Tomcat

### Configuración de una aplicación Spring Boot

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado el **com.javatpoint**.

**Paso 3:** proporcione el Id. Del **artefacto** . Hemos proporcionado el **ejemplo de implementación de spring-boot-war**.

**Paso 4:** agregue la dependencia de **Spring Web** .

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

**Paso 5:** Haga clic en el botón **Generar** . Envuelve toda la especificación relacionada con el proyecto y descarga el archivo **jar** en nuestro sistema local.



**INSTITUTO TÉCNICO  
DE ESTUDIOS  
PROFESIONALES**

Spring Initializr  
Bootstrap your application

Project: Maven Project

Language: Java

Spring Boot: 2.2.2

Project Metadata: Group com.javatpoint, Artifact spring-boot-war-deployment-example

Dependencies: Spring Web (selected)

© 2013-2020 Pivotal Software  
start.spring.io is powered by Spring Initializr and Pivotal Web Services

Generate - Ctrl + A, Explore - Ctrl + Space, Share...

**Paso 6:** extraiga el archivo jar.

**Paso 7: Importe** la carpeta del proyecto mediante los siguientes pasos:

Archivo -> Importar -> Proyecto Maven existente -> Siguiente -> Examinar -> Seleccione la carpeta del proyecto -> Finalizar

Después de importar el proyecto, podemos ver la siguiente estructura de directorios en la sección **Explorador de paquetes** del IDE.





```

└─ spring-boot-war-deployment-example [boot]
    └─ src/main/java
        └─ com.javatpoint
            └─ SpringBootWarDeploymentExampleApplication.java
    └─ src/main/resources
        └─ application.properties
    └─ src/test/java
    └─ JRE System Library [JavaSE-1.8]
    └─ Maven Dependencies
    └─ src
        └─ main
        └─ test
    └─ target
    └─ mvnw
    └─ mvnw.cmd
    └─ pom.xml

```

**Paso 8:** Cree una clase de controlador en el paquete **com.javatpoint**. Hemos creado una clase con el nombre **DemoRestController**.

Dentro de la clase del controlador, hemos definido un método **hello ()** que devuelve un String.

### DemoRestController.java

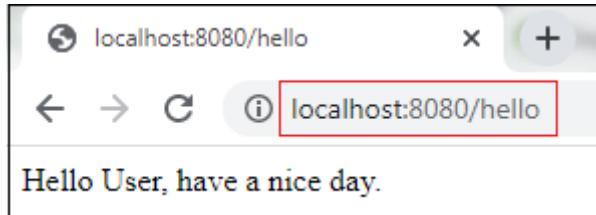
```

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. }

```



**Paso 10:** Abra el navegador e invoque la URL `http://localhost:8080 / hello`.



**Nota:** antes de pasar al siguiente paso, asegúrese de que la aplicación se esté ejecutando correctamente.

## Crear un WAR de Spring Boot

Hace uso del soporte Servlet 3.0 de Spring Framework y nos permite configurar la aplicación cuando el contenedor de servlets la lanza. Hay **tres** pasos para crear un WAR para su implementación:

- Extiende la clase **SpringBootServletInitializer** en la clase principal.
- Marcó el contenedor de servlet integrado como se **proporciona** .
- Actualizar el **JAR de** empaquetado a

Implementemos los tres pasos anteriores en una aplicación.

### Paso

**11:** Abra

el archivo **SpringBootWarDeploymentExampleApplication.java** e inicialice el contexto de servlet requerido por Tomcat. Para lograr lo mismo se extiende la interfaz **SpringBootServletInitializer** .

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

**Paso 12:** anule el método **Configure** .

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
   initializer
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. }
```

**Paso 13:** Abra el archivo **pom.xml** y marque el contenedor de servlets (Tomcat) como se proporciona .

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

**Paso 14:** Necesitamos implementar el archivo **WAR** , así que cambie el tipo de paquete a WAR en el archivo pom.xml.

1. <packaging>war</packaging>





**Paso 15:** Modifique el nombre del archivo WAR final utilizando la etiqueta **<finalName>** para evitar incluir los números de versión. Hemos creado un archivo WAR con el nombre **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>

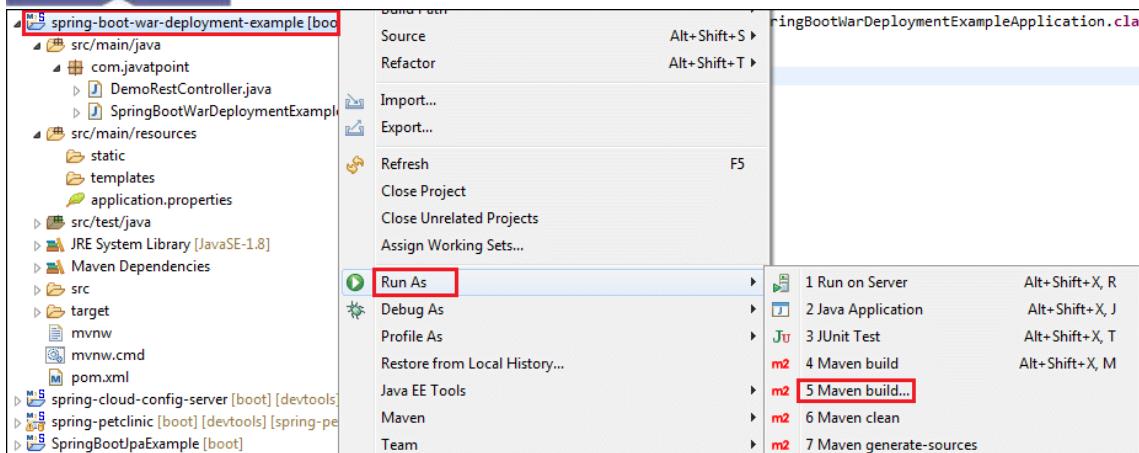
```

Para construir nuestra aplicación WAR implementable en Tomcat, ejecutamos el paquete maven clean. Después de eso, nuestro archivo WAR se genera en **/target/abc.war** (donde se asume **abc** Id de artefacto). Debemos considerar que esta nueva configuración hace que nuestra aplicación Spring Boot sea una aplicación **no independiente**.

**Paso 16:** cree un archivo **WAR** mediante los siguientes pasos:

Haga clic derecho en el proyecto -> Ejecutar como -> 5 Maven Build

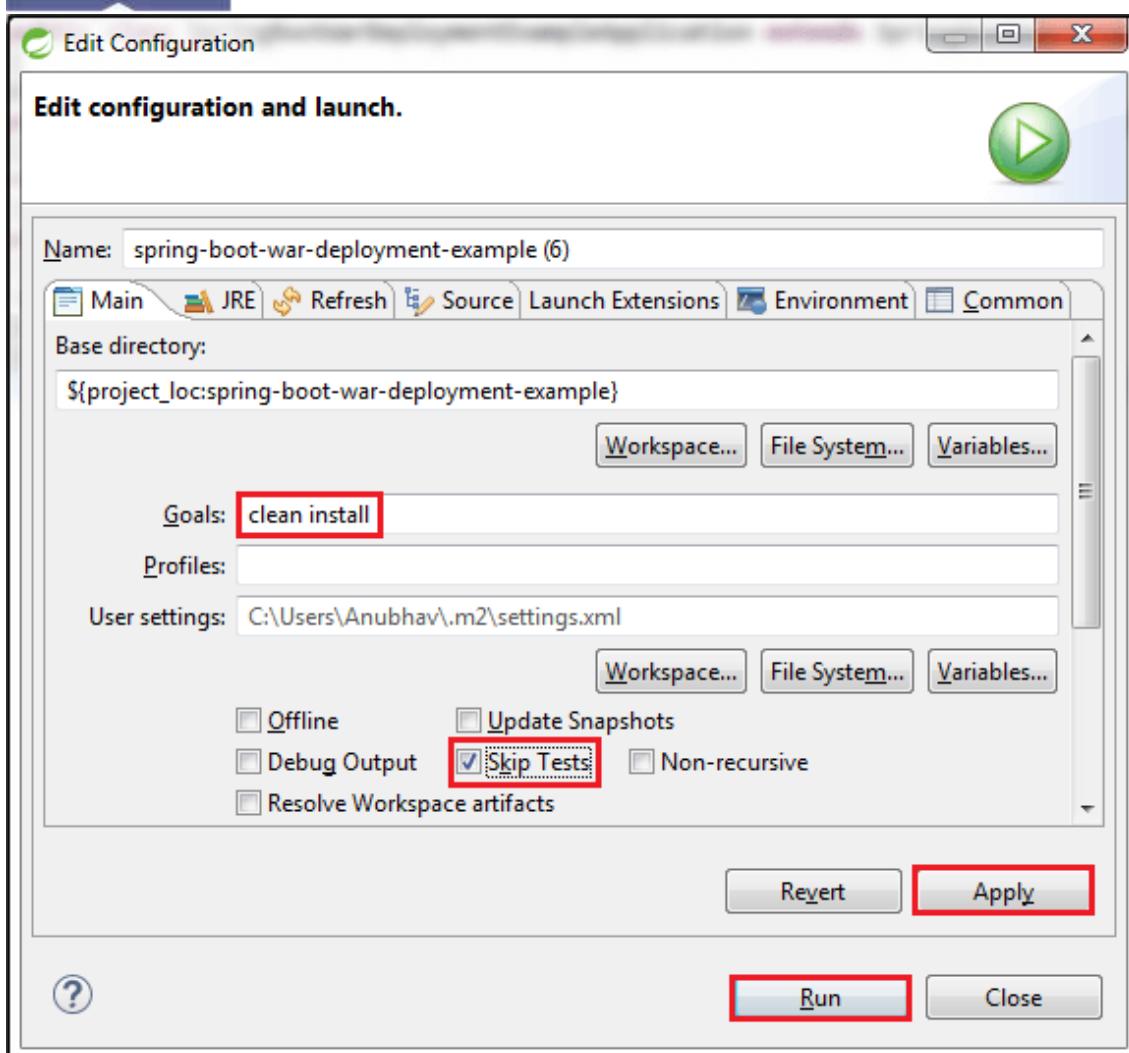




Una **edición de configuración** aparece el cuadro de diálogo en la pantalla.

**Paso 17:** Escriba **instalación limpia** en la etiqueta de **Objetivos** y verifique las **Pruebas de omisión**. Haga clic en el botón **Aplicar** y **Ejecutar**, respectivamente.





Cuando el archivo WAR se crea correctamente, muestra la **ruta del archivo WAR** y un mensaje **BUILD SUCCESS** en la consola, como se muestra en la siguiente figura.

```

Processing war project
Webapp assembled in [3004 msecs]
Building war: C:\Users\Anubhav\Documents\workspace-sts-3.9.9.RELEASE\spring-boot-war-deployment-example\target\web-services.war
--- spring-boot-maven-plugin:2.2.2.RELEASE:repackage (repackage) @ spring-boot-war-deployment-example ---
Replacing main artifact with repackaged archive

--- maven-install-plugin:2.5.2:install (default-install) @ spring-boot-war-deployment-example ---
Installing C:\Users\Anubhav\Documents\workspace-sts-3.9.9.RELEASE\spring-boot-war-deployment-example\target\web-services.war to C:\Us
Installing C:\Users\Anubhav\Documents\workspace-sts-3.9.9.RELEASE\spring-boot-war-deployment-example\pom.xml to C:\Users\Anubhav\.m2\

BUILD SUCCESS
-----
Total time: 41.725 s
Finished at: 2020-01-09T10:59:06+05:30
-----
```

**Paso 18:** Copie la **ruta** y acceda a la carpeta de **destino** de la aplicación. Encontramos el archivo WAR dentro de la carpeta de destino con el



mismo nombre que hemos especificado en el archivo pom.xml. En nuestro caso, la ruta es:

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

classes	1/9/2020 10:58 AM	File folder
generated-sources	1/9/2020 10:58 AM	File folder
maven-archiver	1/9/2020 10:58 AM	File folder
maven-status	1/9/2020 10:58 AM	File folder
web-services	1/9/2020 10:58 AM	File folder
<b>web-services.war</b>	<b>1/9/2020 10:59 AM</b>	<b>WAR File</b>
web-services.war.original	1/9/2020 10:58 AM	ORIGINAL File

## Implementar el archivo WAR en Tomcat

Para implementar el archivo WAR, siga los pasos a continuación:

**Paso 19:** Descargue e instale el **servidor Apache Tomcat**, si no está instalado.

**Paso 20:** Copie el archivo WAR (**web-services.war**) y péguelo dentro de la carpeta **webapps** de Tomcat. En nuestro caso, la ubicación de la carpeta webapps es:

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

docs	1/15/2020 10:19 AM	File folder
examples	1/15/2020 10:19 AM	File folder
host-manager	1/15/2020 10:19 AM	File folder
manager	1/15/2020 10:19 AM	File folder
ROOT	1/15/2020 10:19 AM	File folder
web-services	1/15/2020 10:41 AM	File folder
<b>web-services.war</b>	<b>1/9/2020 10:59 AM</b>	<b>WAR File</b>

**Paso 21:** Ahora abra el símbolo del sistema y escriba los siguientes comandos:

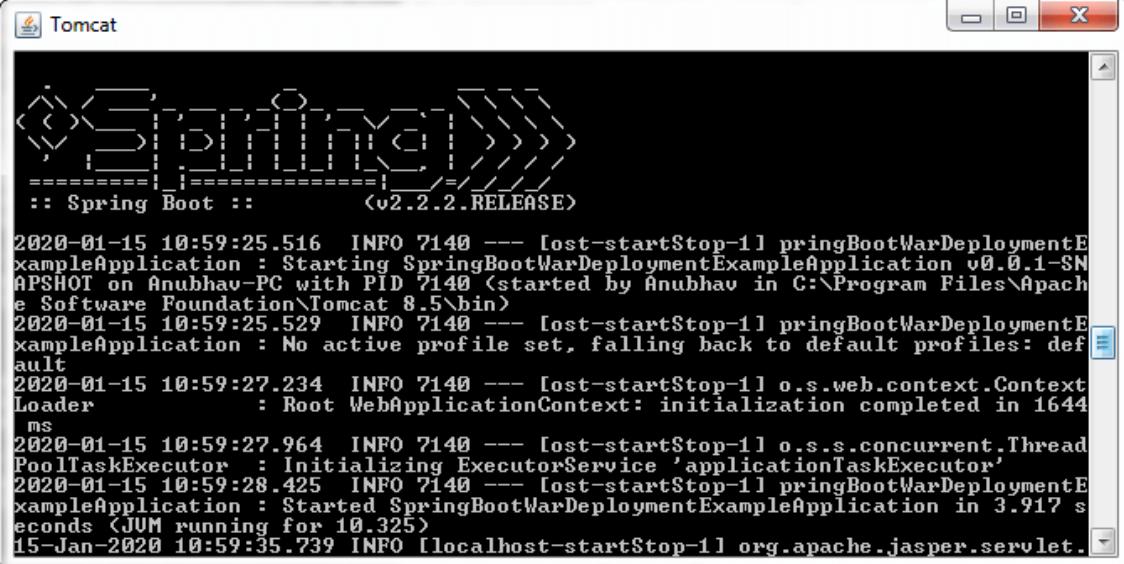
1. C:\Cd Program Files\Apache Software Foundation\Tomcat 8.5\bin
2. C:\Cd Program Files\Apache Software Foundation\Tomcat 8.5\bin>startup



```
cmd Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

C:\Users\Anubhav>cd\
C:\>cd Program Files\Apache Software Foundation\Tomcat 8.5\bin
C:\Program Files\Apache Software Foundation\Tomcat 8.5\bin>startup
Using CATALINA_BASE: "C:\Program Files\Apache Software Foundation\Tomcat 8.5"
Using CATALINA_HOME: "C:\Program Files\Apache Software Foundation\Tomcat 8.5"
Using CATALINA_TMPDIR: "C:\Program Files\Apache Software Foundation\Tomcat 8.5\t
emp"
```

El comando de **inicio** inicia el servidor Tomcat e implementa el archivo WAR, como se muestra a continuación.



```
Tomcat
:: Spring Boot ::      <v2.2.2.RELEASE>

2020-01-15 10:59:25.516  INFO 7140 --- [ost-startStop-1] pingBootWarDeploymentExampleApplication : Starting SpringBootWarDeploymentExampleApplication v0.0.1-SNAPSHOT on Anubhav-PC with PID 7140 (started by Anubhav in C:\Program Files\Apache Software Foundation\Tomcat 8.5\bin)
2020-01-15 10:59:25.529  INFO 7140 --- [ost-startStop-1] pingBootWarDeploymentExampleApplication : No active profile set, falling back to default profiles: default
2020-01-15 10:59:27.234  INFO 7140 --- [ost-startStop-1] o.s.web.context.ContextLoader      : Root WebApplicationContext: initialization completed in 1644 ms
2020-01-15 10:59:27.964  INFO 7140 --- [ost-startStop-1] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-01-15 10:59:28.425  INFO 7140 --- [ost-startStop-1] pingBootWarDeploymentExampleApplication : Started SpringBootWarDeploymentExampleApplication in 3.917 seconds (JVM running for 10.325)
15-Jan-2020 10:59:35.739 INFO [localhost-startStop-1] org.apache.jasper.servlet.
```

La siguiente imagen muestra que WAR se implementó correctamente.

```
15-Jan-2020 10:59:36.202 WARNING [localhost-startStop-1] org.apache.catalina.util.SessionIdGeneratorBase.createSecureRandom Creation of SecureRandom instance for session ID generation using [SHA1PRNG] took [335] milliseconds.
15-Jan-2020 10:59:36.265 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployWAR Deployment of web application archive [C:\Program Files\Apache Software Foundation\Tomcat 8.5\webapps\web-services.war] has finished in [15.521] ms
```

**Paso 23:** Abra el navegador e invoque la URL `http://localhost: 8080 / web-services / hello`. Devuelve el mensaje **Hello User, have a nice day..**

[Download WAR Deployment Example Project](#)





## Spring Boot AOP

La aplicación generalmente se desarrolla con múltiples capas. Una aplicación Java típica tiene las siguientes capas:

- **Capa web:** expone los **servicios** mediante REST o aplicación web.
- **Capa empresarial:** implementa la **lógica empresarial** de una aplicación.
- **Capa de datos:** Implementa la **lógica de persistencia** de la aplicación.

La responsabilidad de cada capa es diferente, pero hay algunos aspectos comunes que se aplican a todas las capas: **registro, seguridad, validación, almacenamiento en caché**, etc. Estos aspectos comunes se denominan **preocupaciones transversales**.

Si implementamos estas preocupaciones en cada capa por separado, el código se vuelve más difícil de mantener. Para superar este problema, la **programación orientada a aspectos** (AOP) proporciona una solución para implementar preocupaciones transversales.

- Implementar la preocupación transversal como aspecto.
- Defina cortes de puntos para indicar dónde se debe aplicar el aspecto.

Garantiza que las preocupaciones transversales se definan en un componente de código cohesivo.

### AOP

**AOP (Programación Orientada a Aspectos)** es un patrón de programación que aumenta la modularidad al permitir la separación de la **preocupación transversal (cross-cutting concern)**. Estas preocupaciones transversales son diferentes de la lógica empresarial principal. Podemos agregar comportamiento adicional al código existente sin modificar el código en sí.

El marco de AOP de Spring nos ayuda a implementar estas preocupaciones transversales.

Usando AOP, definimos la funcionalidad común en un solo lugar. Somos libres de definir cómo y dónde se aplica esta funcionalidad sin modificar la clase a la



que estamos aplicando la nueva característica. La preocupación transversal ahora se puede modular en clases especiales, llamadas **aspect**.

Hay **dos** beneficios de los aspectos:

- Primero, la lógica para cada preocupación ahora está en un lugar en lugar de estar dispersa por todo el código base.
- En segundo lugar, los módulos comerciales solo contienen código para su preocupación principal. La preocupación secundaria se ha trasladado al **aspecto**.

Los aspectos tienen la responsabilidad que se va a implementar, llamados **consejos**. Podemos implementar la funcionalidad de un aspecto en un programa en uno o más puntos de unión.

## Beneficios de AOP

- Está implementado en Java puro.
- No es necesario un proceso de compilación especial.
- Solo admite puntos de unión de ejecución de métodos.
- Solo está disponible el tejido en tiempo de ejecución.
- Hay dos tipos de proxy AOP disponibles: **proxy dinámico JDK** y **proxy CGLIB**.

## Preocupación transversal

La preocupación transversal es una preocupación que queremos implementar en varios lugares de una aplicación. Afecta a toda la aplicación.

## Terminología AOP

- **Aspect:** Un aspecto es un módulo que encapsula **advice** y **pointcuts** y proporciona un aspecto **transversal**. Una aplicación puede tener cualquier número de aspect. Podemos implementar un aspect usando una clase regular anotada con la anotación **@Aspect**.
- **Pointcut:** Un pointcut es una expresión que selecciona uno o más puntos de unión donde se ejecuta el advice. Podemos definir pointcut usando **expresiones** o **patrones**. Utiliza diferentes tipos de expresiones que



coinciden con los puntos de unión. En Spring Framework, se utiliza el lenguaje de expresión de corte de puntos de **AspectJ**.

- **Join point:** Un punto de unión es un punto en la aplicación donde aplicamos un **aspect AOP**. O es una instancia de ejecución específica de un advice. En AOP, el join point puede ser una **ejecución de método, manejo de excepciones, cambio de valor de variable de objeto**, etc.
- **Advice:** El consejo es una acción que tomamos **antes o después de** la ejecución del método. La acción es un fragmento de código que se invoca durante la ejecución del programa. Hay **cinco** tipos de advices en el marco de Spring AOP: **before, after, after-returning, after-throwing, and around advice**. Los advices se toman para un **join point** en particular. Discutiremos estos consejos más a fondo en esta sección.
- **Target object::** un objeto sobre el que se aplican los advices se denomina **Target object**. Los objetos de destino son siempre un **proxy**. Significa que se crea una subclase en tiempo de ejecución en la que se anula el método de destino y se incluyen consejos en función de su configuración.
- **Weaving:** Es un proceso de **vinculación de aspectos** con otros tipos de aplicaciones. Podemos realizar el tejido en **tiempo de ejecución, tiempo de carga y tiempo de compilación**.

**Proxy:** Es un objeto que se crea después de aplicar un advice a un objeto de destino que se llama **proxy**. Spring AOP implementa el **proxy dinámico JDK** para crear las clases de proxy con clases de destino e invocaciones de advices. Se denominan clases de proxy AOP.

## AOP frente a OOP

Las diferencias entre AOP y OOP son las siguientes:

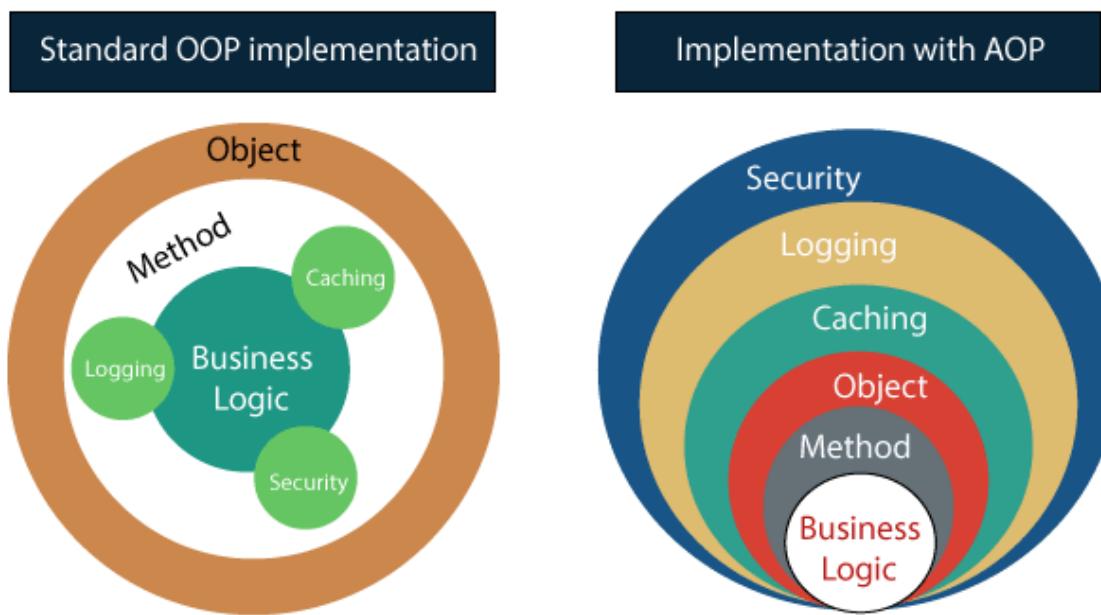
AOP	OOP
<b>Aspect:</b> una unidad de código que encapsula los puntos, los consejos y los atributos.	<b>Clase:</b> una unidad de código que encapsula métodos y atributos.





<b>Pointcut:</b> Define el conjunto de puntos de entrada en los que se ejecuta el advice.	<b>Firma del método:</b> define los puntos de entrada para la ejecución de los cuerpos del método.
<b>Advice:</b> Es una implementación de preocupaciones transversales.	<b>Órganos del método:</b> Es una implementación de las preocupaciones de la lógica empresarial.
<b>Waver:</b> construye código (fuente u objeto) con advices.	<b>Compilador:</b> convierte el código fuente en código objeto.

## Bean in Spring contrainer



## Spring AOP frente a AspectJ

Las diferencias entre AOP y OOP son las siguientes:

Spring AOP	AspectJ





Es necesario un proceso de compilación independiente.	Requiere el compilador AspectJ.
Solo admite pointcuts de ejecución de métodos.	Es compatible con todos los pointcuts.
Se puede implementar en beans administrados por Spring Container.	Se puede implementar en todos los objetos de dominio.
Solo admite tejido a nivel de método.	Puede agitar campos, métodos, constructores, inicializadores estáticos, clase final, etc.

## Tipos de advices de AOP

Hay cinco tipos de advices de AOP que son los siguientes:

- Before Advice
- After Advice
- Around Advice
- After Throwing
- After Returning

**Before advice:** un advice que se ejecuta antes de un join point, se llama antes del advice. Usamos la anotación **@Before** para marcar un before Advice.

**After Advice:** un advice que se ejecuta después de un join point, se llama after advice. Usamos la anotación **@After** para marcar un consejo como after advice.

**Around Advice:** un advice que se ejecuta antes y después de un join point, se llama around advice.

**After Throwing Advice:** un advice que se ejecuta cuando un join point arroja una excepción.

**After Returning Advice:** un advice que se ejecuta cuando un método se ejecuta correctamente.





Antes de implementar el AOP en una aplicación, debemos agregar la dependencia **Spring AOP** en el archivo pom.xml.

## Spring Boot Starter AOP

Spring Boot Starter AOP es una dependencia que proporciona Spring AOP y AspectJ. Donde AOP proporciona capacidades básicas de AOP mientras que AspectJ proporciona un marco AOP completo.

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

En la siguiente sección, implementaremos los diferentes consejos en la aplicación.

## Spring Boot AOP Before Advice

Antes se utiliza el asesoramiento en Programación Orientada a Aspectos para lograr la transversalidad. Es un tipo de aviso que asegura que un aviso se ejecute antes de la ejecución del método. Usamos la anotación **@Before** para implementar el before advice.

Entendamos antes los advices a través de un ejemplo.

### Ejemplo de Spring Boot antes de asesoramiento

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

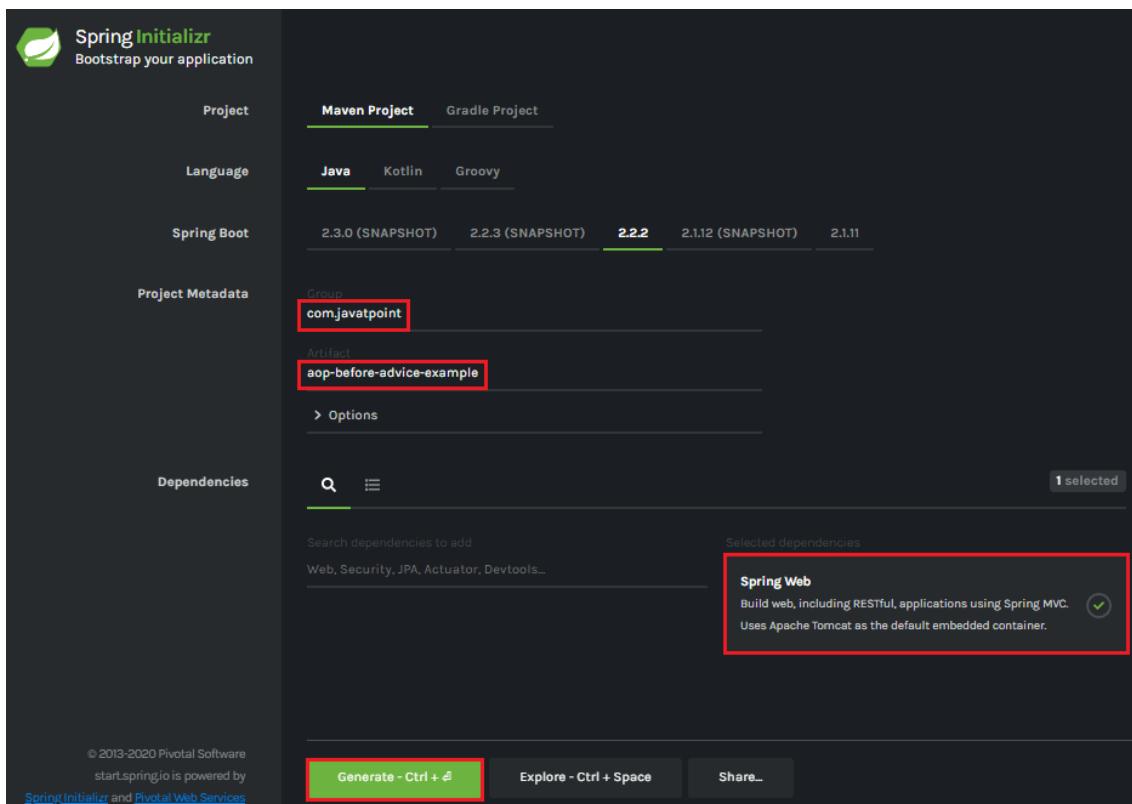
**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado el nombre del grupo **com.javatpoint**.



**Paso 3:** proporcione el **Id. Del artefacto.** Hemos proporcionado el **ejemplo de Artifact Id aop-before-advice.**

**Paso 4:** agregue la dependencia de **Spring Web**.

**Paso 5:** Haga clic en el botón **Generar**. Cuando hacemos clic en el botón Generar, envuelve todas las especificaciones en un archivo **jar** y lo descarga al sistema local.



**Paso 6: extraiga** el archivo jar descargado.

**Paso 7: Importe** la carpeta mediante los siguientes pasos:

Archivo -> Importar -> Proyectos existentes de Maven -> Siguiente -> Examinar la carpeta **aop-before-advice-example** -> Finalizar.

**Paso 8:** Abra el archivo **pom.xml** y agregue la siguiente dependencia de **AOP**. Es un iniciador para la programación orientada a aspectos con **Spring AOP** y **AspectJ**.

### 1. <dependency>



2. <**groupIdgroupId**>
3. <**artifactIdartifactId**>
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.  <modelVersionmodelVersion>
3.  <groupIdgroupId>
4.  <artifactIdartifactId>
5.  <versionversion>
6.  <packagingpackaging>
7.  <namename>
8.  <descriptiondescription>
9.  <parent>
10.   <groupIdgroupId>
11.   <artifactIdartifactId>
12.   <versionversion>
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.     <groupIdgroupId>
23.     <artifactIdartifactId>

```





```

24.      </dependency>
25.  <dependency>
26.    <groupId>org.springframework.boot</groupId>
27.    <artifactId>spring-boot-starter-aop</artifactId>
28.  </dependency>
29. </dependencies>
30.
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>

```

**Paso 9:** Abra el archivo **AopBeforeAdviceExampleApplication.java** y agregue una anotación **@EnableAspectJAutoProxy**.

1. **@EnableAspectJAutoProxy ( proxyTargetClass = true )**

Permite el soporte para el manejo de componentes marcados con la anotación **@Aspect** de AspectJ. Se usa con la anotación **@Configuration**. Podemos controlar el tipo de proxy utilizando el atributo **proxyTargetClass**. Su valor predeterminado es **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. }
```

**Paso 10:** Cree un paquete con el nombre **com.javatpoint.model**.

**Paso 11:** Cree una clase de modelo en el paquete **com.javatpoint.model**. Hemos creado una clase con el nombre **Employee**. En la clase, defina lo siguiente:

- Defina tres variables **emplId**, **firstName** y **secondName** de tipo String.
- Genere **Getters y Setters**.
- Crear un **valor default**

### **Employee.java**

```

1. package com.javatpoint.model;
2. public class Employee
3. {
4.     private String empld;
5.     private String firstName;
6.     private String secondName;
7.     //default constructor
8.     public Employee()
9.     {
10.    }
11.    public String getEmpld()
12. {
```





```

13. return empld;
14. }
15. public void setEmpld(String empld)
16. {
17. this.empld = empld;
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. }
```

**Paso 12:** Cree un paquete con el nombre **com.javatpoint.controller**.

**Paso 13:** Cree una clase de controlador en el paquete **com.javatpoint.controller**. Hemos creado una clase con el nombre **EmployeeController**.

En la clase de controlador, hemos definido las dos asignaciones, una para agregar un empleado y la otra para eliminar a un empleado.

### **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("emplId") String emplId, @RequestParam("firstName") String firstName, @RequestParam("secondName") String secondName)
16.     {
17.         return employeeService.createEmployee(emplId, firstName, secondName);
18.     }
19.     @RequestMapping(value = "/remove/employee", method = RequestMethod.GET)
20.     public String removeEmployee( @RequestParam("emplId") String emplId)
21.     {
22.         employeeService.deleteEmployee(emplId);
23.         return "Employee removed";
24.     }
25. }
```

**Paso 14:** Cree un paquete con el nombre **com.javatpoint.service**.

**Paso 15:** Cree una clase de servicio en el paquete **com.javatpoint.service**. Hemos creado una clase con el nombre **EmployeeService**.





En la clase Service, hemos definido dos métodos **createEmployee** y **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 empld, String fname, String sname)
8.     {
9.         Employee emp = new Employee();
10.        emp.setEmpld(empld);
11.        emp.setFirstName(fname);
12.        emp.setSecondName(sname);
13.        return emp;
14.    }
15.    public void deleteEmployee(String empld)
16.    {
17.    }
18. }
```

**Paso 16:** Cree un paquete con el nombre **com.javatpoint.aspect**.

**Paso 17:** Cree una clase de aspect en el paquete **com.javatpoint.aspect**. Hemos creado una clase con el nombre **EmployeeServiceAspect**.

En la clase de aspect, hemos definido la lógica de los before advices.

### **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(empld, fname, sname)")
11.     public void beforeAdvice(JoinPoint joinPoint, String empld, 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 - "
+ empld);
14.     }
15. }

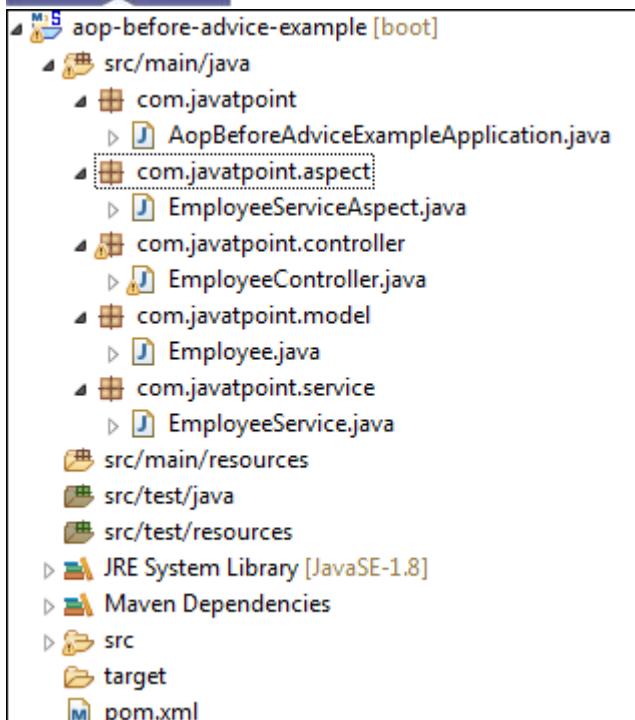
```

En la clase anterior:

- **execution(expresión):** La expresión es un método sobre el que se debe aplicar un advice.
- **@Before:** Marca una función como un advice a ejecutar antes del método que cubre PointCut.

Después de crear todos los módulos, el directorio del proyecto tiene el siguiente aspecto:





Hemos configurado todos los módulos. Ahora ejecutaremos la aplicación.

**Paso 18:** Abra el archivo e **AopBeforeAdviceExampleApplication.java** y ejecútelo como aplicación Java.

**Paso 19:** Abra el navegador e invoque la siguiente URL: `http://localhost: 8080 / add / employee? EmpId = {id} & firstName = {fname} & secondName = {sname}`

En la URL anterior, **/ add / employee** es el mapeo que hemos creado en la clase Controller. Hemos utilizado dos separadores (**?**) Y (**&**) para separar dos valores.



En la salida anterior, hemos asignado **emId 101**, **firstName = Tim** y **secondName = cook**.

Echemos un vistazo a la consola. Vemos que antes de invocar el método **createEmployee ()** de la clase **EmployeeService**, el

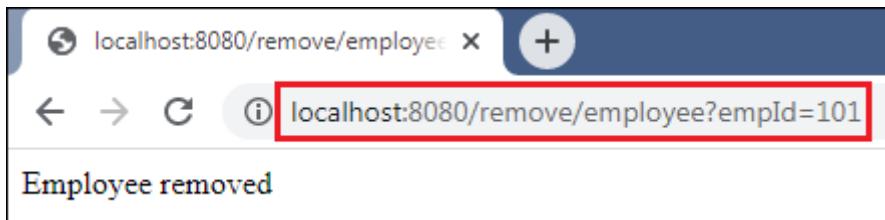




método **beforeAdvice ()** de la clase **EmployeeServiceAspect** invoca, como se muestra a continuación.

```
2020-01-13 16:31:54.730 INFO 4192 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : 
Before method:Employee com.javatpoint.service.EmployeeService.createEmployee(String,String,String)
Creating Employee with first name - Tim, second name - Cook and id - 101
```

Del mismo modo, también podemos eliminar a un empleado invocando la URL `http://localhost: 8080 / remove / employee? EmpId = 101`. Devuelve un mensaje **Empleado eliminado**, como se muestra en la siguiente figura.



En esta sección, hemos aprendido el funcionamiento de los before advices. En la siguiente sección, aprenderemos el funcionamiento de los after advices y lo implementaremos en una aplicación.

[Descargar proyecto de ejemplo de AOP Before Advice](#)

## Spring Boot AOP After Advice

El asesoramiento posterior se utiliza en la programación orientada a aspectos para lograr la transversalidad. Es un tipo de aviso que asegura que un aviso se ejecute después de la ejecución del método. Usamos la anotación **@After** para implementar el after advice.

Entendamos after advice a través de un ejemplo.

### Spring Boot after advice en un ejemplo

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

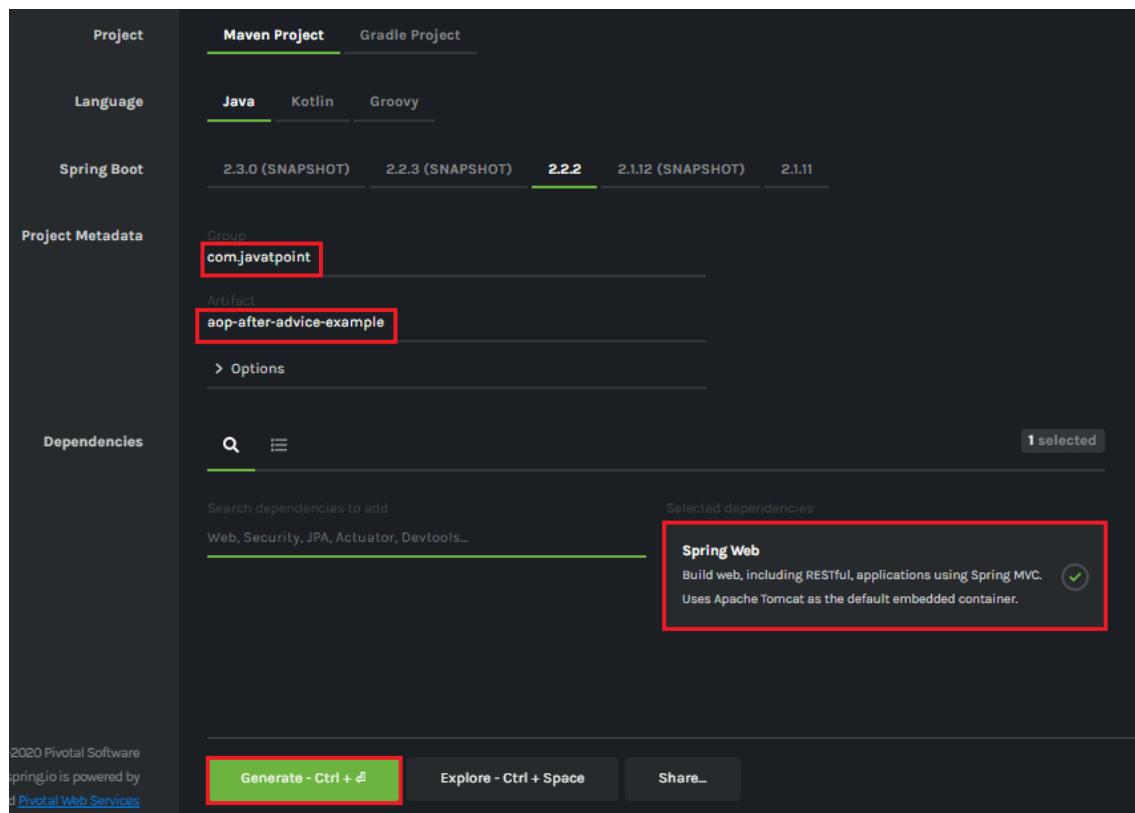
**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado el nombre del grupo **com.javatpoint**.



**Paso 3:** proporcione el **Id. Del artefacto.** Hemos proporcionado el Artifact Id **aop-after-advice-example**.

**Paso 4:** agregue la dependencia de **Spring Web**.

**Paso 5:** Haga clic en el botón **Generar**. Cuando hacemos clic en el botón Generar, envuelve todas las especificaciones en un archivo **jar** y lo descarga al sistema local.



**Paso 6:** extraiga el archivo jar descargado.

**Paso 7: Importe** la carpeta mediante los siguientes pasos:

Archivo -> Importar -> Proyectos existentes de Maven -> Siguiente -> Examinar la carpeta **aop-after-advice-example** -> Finalizar.

**Paso 8:** Abra el archivo **pom.xml** y agregue la siguiente dependencia de **AOP**. Es un iniciador para la programación orientada a aspectos con **Spring AOP** y **AspectJ**.

1. <**dependency**>



2. <**groupIdgroupId**>
3. <**artifactIdartifactId**>
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. <**modelVersionmodelVersion**>
3. <**groupIdgroupId**>
4. <**artifactIdartifactId**>
5. <**versionversion**>
6. <**packagingpackaging**>
7. <**namename**>
8. <**descriptiondescription**>
9. <**parent**>
10. <**groupIdgroupId**>
11. <**artifactIdartifactId**>
12. <**versionversion**>
13. <**relativePath** /> <!-- lookup parent from repository -->
14. </**parent**>
15. <**properties**>
16. <**project.build.sourceEncodingproject.build.sourceEncoding**>
17. <**project.reporting.outputEncodingproject.reporting.outputEncoding**>
18. <**java.versionjava.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>
30.
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>
```

**Paso 9:** Abra el archivo **AopAfterAdviceExampleApplication.java** y agregue una anotación **@EnableAspectJAutoProxy**.

1. **@EnableAspectJAutoProxy ( proxyTargetClass = true )**

Permite el soporte para el manejo de componentes marcados con la anotación **@Aspect** de AspectJ. Se usa con la anotación **@Configuration**. Podemos controlar el tipo de proxy utilizando el atributo **proxyTargetClass**. Su valor predeterminado es **falso**.

### **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. }
```

**Paso 10:** Cree un paquete con el nombre **com.javatpoint.model**.

**Paso 11:** Cree una clase de modelo en el paquete **com.javatpoint.model**. Hemos creado una clase con el nombre **Employee**. En la clase, defina lo siguiente:

- Defina tres variables **emplId**, **firstName** y **secondName** de tipo String.
- Genere **Getters y Setters**.
- Crear un **valor predeterminado**

### **Employee.java**

```

1. package com.javatpoint.model;
2. public class Employee
3. {
4.     private String emplId;
5.     private String firstName;
6.     private String secondName;
7.     //default constructor
8.     public Employee()
9.     {
10.    }
11.    public String getEmplId()
12.    {
13.        return emplId;
14.    }
15.    public void setEmplId(String emplId)
```





```

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. }
```

**Paso 12:** Cree un paquete con el nombre **com.javatpoint.controller**.

**Paso 13:** Cree una clase de controlador en el paquete **com.javatpoint.controller**. Hemos creado una clase con el nombre **EmployeeController**.

En la clase de controlador, hemos definido las dos asignaciones, una para agregar un empleado y la otra para eliminar a un empleado.

### 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("emplId") String emplId, @RequestPara
m("firstName") String firstName, @RequestParam("secondName") String secondName)
16.     {
17.         return employeeService.createEmployee(emplId, firstName, secondName);
18.     }
19.     @RequestMapping(value = "/remove/employee", method = RequestMethod.GET)
20.     public String removeEmployee( @RequestParam("emplId") String emplId)
21.     {
22.         employeeService.deleteEmployee(emplId);
23.         return "Employee removed";
24.     }
25. }
```

#### Paso 14: Cree un paquete con el nombre **com.javatpoint.service**.

**Paso 15:** Cree una clase de servicio en el paquete **com.javatpoint.service**. Hemos creado una clase con el nombre **EmployeeService**.

En la clase Service, hemos definido dos métodos **createEmployee** y **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 empld, String fname, String sna
    me)
8.     {
9.         Employee emp = new Employee();
10.        emp.setEmpld(empld);
11.        emp.setFirstName(fname);
12.        emp.setSecondName(sname);
13.        return emp;
14.    }
15.    public void deleteEmployee(String empld)
16.    {
17.    }
18. }
```

**Paso 16:** Cree un paquete con el nombre **com.javatpoint.aspect**.

**Paso 17:** Cree una clase de aspecto en el paquete **com.javatpoint.aspect**. Hemos creado una clase con el nombre **EmployeeServiceAspect**.

En la clase de aspecto, hemos definido la lógica posterior al consejo.

### 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(emplId, fname, sname)")
11. public void afterAdvice(JoinPoint joinPoint, String emplId, String fname, S
    tring sname) {
12. System.out.println("After method:" + joinPoint.getSignature());
13. System.out.println("Creating Employee with first name - " + fname + ", se
    cond name - " + sname + " and id - " + emplId);
14. }
15. }
```

En la clase anterior:

- **execution (expresión):** La expresión es un método sobre el que se debe aplicar un advice.
- **@After:** el método anotado con **@After** se ejecuta después de todos los métodos que coinciden con la expresión pointcut.

Después de crear todos los módulos, el directorio del proyecto tiene el siguiente aspecto:





```

M S aop-after-advice-example [boot]
  ▾ src/main/java
    ▾ com.javatpoint
      ▷ AopAfterAdviceExampleApplication.java
    ▾ com.javatpoint.aspect
      ▷ EmployeeServiceAspect.java
    ▾ com.javatpoint.controller
      ▷ EmployeeController.java
    ▾ com.javatpoint.model
      ▷ Employee.java
    ▾ com.javatpoint.service
      ▷ EmployeeService.java
  ▾ src/main/resources
  ▾ src/test/java
  ▾ src/test/resources
  ▷ JRE System Library [JavaSE-1.8]
  ▷ Maven Dependencies
  ▷ src
  ▾ target
  pom.xml

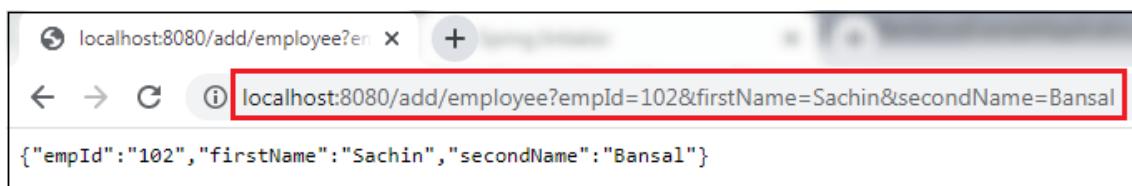
```

Hemos configurado todos los módulos. Ahora ejecutaremos la aplicación.

**Paso 18:** Abra el archivo **AopAfterAdviceExampleApplication.java** y ejecútelo como aplicación Java.

**Paso 19:** Abra el navegador e invoque la siguiente URL: *http://localhost: 8080 / add / employee? Empld = {id} & firstName = {fname} & secondName = {sname}*

En la URL anterior, **/ add / employee** es el mapeo que hemos creado en la clase Controller. Hemos utilizado dos separadores (**?**) Y (**&**) para separar dos valores.



En la salida anterior, hemos asignado **emId 102**, **firstName = Sachin** y **secondName = Bansal**.

Echemos un vistazo a la consola. Vemos que después de invocar el método **createEmployee ()** de la clase **EmployeeService**, el

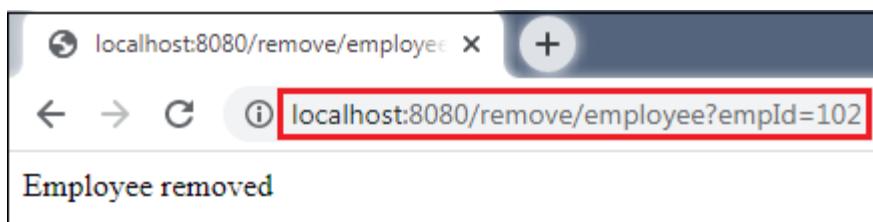




método **afterAdvice ()** de la clase **EmployeeServiceAspect** invoca, como se muestra a continuación.

```
2020-01-13 17:14:49.029 INFO 6168 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet
After method:Employee com.javatpoint.service.EmployeeService.createEmployee(String,String,String)
Successfully created Employee with first name - Sachin, second name - Bansal and id - 102
```

Del mismo modo, también podemos eliminar a un empleado invocando la URL `http://localhost: 8080 / remove / employee? Empld = 102`. Devuelve un mensaje **Empleado eliminado**, como se muestra en la siguiente figura.



En esta sección, hemos aprendido el funcionamiento de los after advice. En la siguiente sección, aprenderemos el funcionamiento de los around advice.

[Descargar proyecto de ejemplo de AOP After Advice](#)

## Spring Boot AOP Around Advice

El consejo de Around está representado por la anotación **@Around**. Se ejecuta antes y después de un punto de unión. Es el consejo más poderoso. También proporciona más control para que el usuario final pueda negociar con **ProceedingJoinPoint**.

Implementemos los consejos en una aplicación.

### Ejemplo de consejo de Spring Boot Around

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado el nombre del grupo **com.javatpoint**.





**Paso 3:** proporcione el **Id. Del artefacto**. Hemos proporcionado el **ejemplo de un consejo de consulta de Artifact Id**.

**Paso 4:** agregue la dependencia de **Spring Web**.

**Paso 5:** Haga clic en el botón **Generar**. Cuando hacemos clic en el botón Generar, envuelve todas las especificaciones en un archivo **jar** y lo descarga al sistema local.

The screenshot shows the configuration interface for a Maven Project. Key settings include:

- Project:** Maven Project (selected)
- Language:** Java (selected)
- Spring Boot:** 2.2.2 (selected)
- Project Metadata:**
  - Group: com.javatpoint
  - Artifact: aop-around-advice-example
- Dependencies:**
  - Search dependencies to add: Web, Security, JPA, Actuator, Devtools...
  - Selected dependencies: Spring Web (highlighted with a red box)
- Buttons:** Generate - Ctrl + F (highlighted with a red box), Explore - Ctrl + Space, Share...

**Paso 6: extraiga** el archivo jar descargado.

**Paso 7: Importe** la carpeta mediante los siguientes pasos:

Archivo -> Importar -> Proyectos existentes de Maven -> Siguiente -> Examinar la carpeta **aop-around-advice-example** -> Finalizar.

**Paso 8:** Abra el archivo **pom.xml** y agregue la siguiente dependencia de **AOP**. Es un iniciador para la programación orientada a aspectos con **Spring AOP y 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>

```

**Paso 9:** cree un paquete con el nombre **com.javatpoint.service**.

**Paso 10:** Cree una clase en el paquete anterior con el nombre **BankService** .

En esta clase, hemos definido un método llamado **displayBalance ()**. Comprueba el número de cuenta. Si el número de cuenta coincide, devuelve el monto total; de lo contrario, devuelve un mensaje.





## 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. }
```

**Paso 11:** cree otro paquete con el nombre **com.javatpoint.aspect**.

**Paso 12:** Cree una clase en el paquete anterior con el nombre **BankAspect**.

En la siguiente clase, hemos definido dos métodos llamados **logDisplayingBalance ()** y método **aroundAdvice ()** .

## 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 invocation of the
   method " + jp.getSignature().getName() + " method");
22. try
23. {
24. jp.proceed();
25. }
26. finally
27. {
28.
29. }
30. System.out.println("The method aroundAdvice() after invocation of the m
   ethod " + jp.getSignature().getName() + " method");
31. }
32. }
```





**Paso 13:** Abra el archivo **AopAroundAdviceExampleApplication.java** y agregue una anotación **@EnableAspectJAutoProxy**.

La anotación permite el soporte para el manejo de componentes marcados con la anotación **@Aspect** de AspectJ . Se usa con la anotación **@Configuration**.

**ConfigurableApplicationContext** es una interfaz que proporciona facilidades para configurar un contexto de aplicación además de los métodos de cliente de contexto de aplicación en 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. ;
7. import com.javatpoint.service.BankService;
8. //@EnableAspectJAutoProxy annotation enables support for handling the
9. components marked with @Aspect annotation. It is similar to tag in the x
10. ml configuration.
11. @EnableAspectJAutoProxy
12. public class AopAroundAdviceExampleApplication
13. {
14.     ConfigurableApplicationContext context = SpringApplication.run(AopAro
15. undAdviceExampleApplication.class, args);
16.     // Fetching the employee object from the application context.
17.     BankService bank = context.getBean(BankService.class);
18.     // Displaying balance in the account
19.     String accnumber = "12345";
20.     bank.displayBalance(accnumber);
21.     // Closing the context object

```





```

21. context.close();
22. }
23. }
```

Después de crear todos los paquetes y clases, el directorio del proyecto se ve así:

```

aop-around-advice-example [boot]
  src/main/java
    com.javatpoint
      AopAroundAdviceExampleApplication.java
    com.javatpoint.aspect
      BankAspect.java
    com.javatpoint.service
      BankService.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
    com.javatpoint
      AopAroundAdviceExampleApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
    target
    HELP.md
    mvnw
    mvnw.cmd
  pom.xml
```

Ahora, ejecute la aplicación.

**Paso 14:** Abra **AopAroundAdviceExampleApplication.java** y ejecútelo como aplicación Java.

```

The method aroundAdvice() before invocation of the method displayBalance method
Inside displayBalance() method
Total balance: 10,000
The method aroundAdvice() after invocation of the method displayBalance method
2020-01-14 14:11:03.685  INFO 3812 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'applicationTaskExecutor'
```

En la salida anterior, vemos que el método `aroundAdvice ()` invoca dos veces. Primero, antes de la ejecución del método **displayBalance ()** y segundo, después de la ejecución del método **displayBalance ()**.



[Descargar proyecto de ejemplo de AOP Around Advice](#)

## Spring Boot AOP After Returning Advice

**After returning** es un advice en Spring AOP que invoca después de la ejecución del join point completo (ejecutar) normalmente. No se invoca si se lanza una excepción. Podemos implementar después de devolver un advice en una aplicación usando la anotación **@AfterReturning**. La anotación marca una función como un advice que debe ejecutarse antes del método cubierto por PointCut.

Después de devolver el advice, se ejecuta cuando la ejecución de un método coincidente devuelve un valor normalmente. El nombre que definimos en el atributo de retorno debe corresponder al nombre de un parámetro en el método de advice. Cuando un método devuelve un valor, el valor se pasará al método de advice como el valor de argumento correspondiente.

Implementemos el after returning advice en una aplicación.

### Spring Boot ejemplo de After Returning Advice

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado el nombre del grupo **com.javatpoint**.

**Paso 3:** proporcione el **Id. Del artefacto**. Hemos proporcionado el Id de artefacto **aop-after-return-advice-example**.

**Paso 4:** agregue la dependencia de **Spring Web** .

**Paso 5:** Haga clic en el botón **Generar** . Cuando hacemos clic en el botón Generar, envuelve todas las especificaciones en un archivo **jar** y lo descarga al sistema local.

**Paso 6:** extraiga el archivo jar descargado.



**Paso 7: Importe** la carpeta mediante los siguientes pasos:

Archivo -> Importar -> Proyectos de Maven existentes -> Siguiente -> Examinar la carpeta **aop-after-return-advice-example** -> Finalizar.

**Paso 8:** Abra el archivo **pom.xml** y agregue la siguiente dependencia de **AOP**. Es un iniciador para la programación orientada a aspectos con **Spring AOP** y **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>
```

**Paso 9:** Crear un paquete con el nombre **com.javatpoint.model** en la carpeta / **java src / main .**

**Paso 10:** Cree una clase con el nombre **Account** en el paquete **com.javatpoint.model.**

En la clase **Account**, haga lo siguiente:

- Se definieron dos variables **accountNumber** y **accountType** de tipo String.



- Haga clic derecho en el archivo -> Fuente -> Generar constructor usando campos
- Generar-getters.  
Haga clic derecho en el archivo -> Fuente -> Generar Getters y Setters -> Seleccionar Getters -> Generar
- Genere un **toString()**  
Haga clic con el botón derecho en el archivo -> Fuente -> Generar 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. }
```

**Paso 11:** cree otro paquete con el nombre **com.javatpoint.service.impl**.

**Paso 12:** En este paquete, cree una clase con el nombre **AccountServiceImpl**.

En esta clase, hemos definido el servicio de cuenta.

### **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. }
```

**Paso 13:** Cree una interfaz con el nombre **AccountService** en el paquete **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. }
```





**Paso 14:** Cree un paquete con el nombre **com.javatpoint.aspect**.

**Paso 15:** Cree una clase con el nombre **AccountAspect** en el paquete **com.javatpoint.aspect**.

En esta clase, hemos implementado el advice **after return** usando la anotación **@AfterReturning**. También hemos definido un método **afterReturningAdvice ()**.

**Nota:** El nombre (account) que definimos en el atributo de **returning** debe corresponder al nombre de un parámetro en el **método de advice**.

### **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.Account
   ServiceImpl.*(..))",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. }
```





**Paso**

**16:** Abra

el archivo **AopAfterReturningAdviceExampleApplication.java** y agregue una anotación **@EnableAspectJAutoProxy**.

La anotación permite el soporte para el manejo de componentes marcados con la anotación **@Aspect** de AspectJ . Se usa con la anotación **@Configuration**.

Hemos utilizado el atributo **proxyTargetClass** de la anotación **@EnableAspectJAutoProxy**. El atributo **proxyTargetClass = true** nos permite usar **proxies CGLIB** (Biblioteca de generación de código) en lugar del enfoque de proxy JDK basado en interfaz predeterminado.

**ConfigurableApplicationContext** es una interfaz que proporciona facilidades para configurar un contexto de aplicación además de los métodos de cliente de contexto de aplicación en 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. ;
7. import com.javatpoint.model.Account;
8. import com.javatpoint.service.impl.AccountService;
9. import com.javatpoint.service.impl.AccountServiceImpl;
10. @SpringBootApplication
11. // @EnableAspectJAutoProxy annotation enables support for handling the
12. // components marked with @Aspect annotation. It is similar to tag in the x
13. // ml configuration.
14. @EnableAspectJAutoProxy(proxyTargetClass=true)
15. public class AopAfterReturningAdviceExampleApplication
16. {
17.     public static void main(String[] args)
18. }
```





```
16. ConfigurableApplicationContext ac = SpringApplication.run(AopAfterRet
   urningAdviceExampleApplication.class, args);
17. //Fetching the account object from the application context
18. AccountService accountService = ac.getBean("accountServiceImpl", Acco
   untServiceImpl.class);
19. Account account;
20. try
21. {
22. account = accountService.getAccountByCustomerId("K2434567");
23. if(account != null)
24. System.out.println(account.getAccountNumber() + "\t" + account.getAccou
   ntType());
25. }
26. catch (Exception e)
27. {
28. System.out.println(e.getMessage());
29. }
30. }
31. }
```

Después de crear todas las clases y paquetes, el directorio del proyecto tiene el siguiente aspecto:





```

M S aop-after-returning-advice-example [boot]
  ▾ src/main/java
    ▾ com.javatpoint
      ▾ AopAfterReturningAdviceExampleApplication.java
    ▾ com.javatpoint.aspect
      ▾ AccountAspect.java
    ▾ com.javatpoint.model
      ▾ Account.java
    ▾ com.javatpoint.service.impl
      ▾ AccountService.java
      ▾ AccountServiceImpl.java
  ▾ src/main/resources
    application.properties
  ▾ src/test/java
  ▾ JRE System Library [JavaSE-1.8]
  ▾ Maven Dependencies
  ▾ src
    target
    mvnw
    mvnw.cmd
  pom.xml

```

### Paso

**17:** Abra el archivo **AopAfterReturningAdviceExampleApplication.java** y ejecútelo como aplicación Java. Muestra la salida, como se muestra a continuación:

```

After Returing method:Account com.javatpoint.service.impl.AccountServiceImpl.getAccountByCustomerId(String)
Account [accountNumber=10863554577, accountType=Current Account]
10863554577 Current Account

```

En la siguiente sección, lo entenderemos después after throwing advice.

[Descargar AOP después de devolver el proyecto de ejemplo de asesoramiento](#)

## Spring Boot AOP After Throwing Advice

Después de lanzar es un tipo de advice en Spring AOP. Garantiza que se ejecute un aviso si un método arroja una excepción. Usamos la anotación **@AfterThrowing** para implementar el advice de after throwing.

**Sintaxis:**



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

Dónde:

**PointCut:** Selecciona una función.

**execution (expresión):** Es una expresión sobre la que se debe aplicar un consejo.

**throwing:** el nombre de la excepción que se devolverá.

## Spring Boot ejemplo de After Throwing Advice

Usaremos el ejemplo anterior en esta sección. Puedes descargar el proyecto o hacer algunas modificaciones en el ejemplo anterior.

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado el nombre del grupo **com.javatpoint**.

**Paso 3:** proporcione el **Id. Del artefacto**. Hemos proporcionado el Artifact Id **aop-after-throwing-advice-example**.

**Paso 4:** agregue la dependencia de **Spring Web** .

**Paso 5:** Haga clic en el botón **Generar** . Cuando hacemos clic en el botón Generar, envuelve todas las especificaciones en un archivo **jar** y lo descarga al sistema local.

**Paso 6:** extraiga el archivo jar descargado.

**Paso 7: Importe** la carpeta mediante los siguientes pasos:

Archivo -> Importar -> Proyectos existentes de Maven -> Siguiente -> Examinar la carpeta **aop-after-throwing-advice-example** -> Finalizar.

**Paso 8:** Abra el archivo **pom.xml** y agregue la siguiente dependencia de **AOP** . Es un iniciador para la programación orientada a aspectos con **Spring AOP** y **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. <**modelVersionmodelVersion**>
5. <**groupIdgroupId**>
6. <**artifactIdartifactId**>
7. <**versionversion**>
8. <**packagingpackaging**>
9. <**namename**>
10. <**descriptiondescription**>
11. <**parent**>
12. <**groupIdgroupId**>
13. <**artifactIdartifactId**>
14. <**versionversion**>
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.versionjava.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>

```

**Paso 9:** Crear un paquete con el nombre **com.javatpoint.model** en / java src / main carpeta .

**Paso 10:** Cree una clase con el nombre **Cuenta** en el paquete **com.javatpoint.model**.

En la clase Cuenta, haga lo siguiente:

- Se definieron dos variables **accountNumber** y **accountType** de tipo String.
- Haga clic derecho en el archivo -> Fuente -> Generar constructor usando campos





- Generar getters.  
Haga clic derecho en el archivo -> Fuente -> Generar Getters y Setters -> Seleccionar Getters -> Generar
- Genere un **toString()**  
Haga clic con el botón derecho en el archivo -> Fuente -> Generar 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
24.                  =" + accountType + "]";
24.    }

```





25. }

**Paso 11:** cree otro paquete con el nombre **com.javatpoint.service.impl**.

**Paso 12:** En este paquete, cree una clase con el nombre **AccountServiceImpl**.

En esta clase, hemos definido el servicio de cuenta.

### **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 Exc
eption
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. }
```

**Paso 13:** Cree una interfaz con el nombre **AccountService** en el paquete **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. }
```

**Paso 14:** Cree un paquete con el nombre **com.javatpoint.aspect**.





**Paso 15:** Cree una clase con el nombre **AccountAspect** en el paquete **com.javatpoint.aspect**.

En esta clase, hemos implementado el consejo after throwing usando la anotación **@AfterThrowing**. También hemos definido un método **afterThrowingAdvice ()**.

**Nota:** El nombre (ex) que definimos en el atributo throwing debe corresponder al nombre de un parámetro en el método advice. De lo contrario, los advices no se ejecutarán.

### 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.Account
   ServiceImpl.*(..))",throwing="ex")
12. public void afterThrowingAdvice(JoinPoint joinPoint, Exception ex)
13. {
14. System.out.println("After Throwing exception in method:"+joinPoint.getSi
   gnature());
15. System.out.println("Exception is:"+ex.getMessage());
16. }
17. }
```

**Paso 16:** Abra el archivo **AopAfterThrowingAdviceExampleApplication.java** y agregue una anotación **@EnableAspectJAutoProxy**.



La anotación permite el soporte para el manejo de componentes marcados con la anotación **@Aspect** de AspectJ . Se usa con la anotación @Configuration.

Hemos utilizado el atributo **proxyTargetClass** de la anotación **@EnableAspectJAutoProxy**. El atributo **proxyTargetClass = true** nos permite usar **proxies CGLIB** (Biblioteca de generación de código) en lugar del enfoque de proxy JDK basado en interfaz predeterminado.

**ConfigurableApplicationContext** es una interfaz que proporciona facilidades para configurar un contexto de aplicación además de los métodos de cliente de contexto de aplicación en 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 x
    ml configuration.
11. @EnableAspectJAutoProxy(proxyTargetClass=true)
12. public class AopAfterThrowingAdviceExampleApplication
13. {
14.     public static void main(String[] args)
15.     {
16.         ConfigurableApplicationContext ac = SpringApplication.run(AopAfterThr
            owingAdviceExampleApplication.class, args);
17.         // Fetching the account object from the application context

```





```
18. AccountService accountService = ac.getBean("accountServiceImpl", Acco
untServiceImpl.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.getAccou
ntType());
26. }
27. catch (Exception e)
28. {
29. System.out.println(e.getMessage());
30. e.printStackTrace();
31. }
32. }
33. }
```

Después de crear todas las clases y paquetes, el directorio del proyecto tiene el siguiente aspecto:





```

aop-after-throwing-advice-example [boot]
├── src/main/java
│   ├── com.javatpoint
│   │   ├── AopAfterThrowingAdviceExampleApplication.java
│   ├── com.javatpoint.aspect
│   │   └── AccountAspect.java
│   ├── com.javatpoint.model
│   │   └── Account.java
│   └── com.javatpoint.service.impl
│       ├── AccountService.java
│       └── AccountServiceImpl.java
├── src/main/resources
└── src/test/java
    └── JRE System Library [JavaSE-1.8]
        └── Maven Dependencies
└── src
    ├── target
    ├── mvnw
    ├── mvnw.cmd
    └── pom.xml

```

**Paso 17:** Abra el archivo **AopAfterThrowingAdviceExampleApplication.java** y ejecútelo como aplicación Java. Muestra la salida, como se muestra a continuación:

```

After Throwing exception in method:Account com.javatpoint.service.impl.AccountServiceImpl.getAccountByCustomerId(String)
Exception is:Invalid! Customer Id
Invalid! Customer Id

```

[Descargar el proyecto de ejemplo de AOP después de lanzar un consejo](#)

Seguir por:

## Spring Boot JPA

### ¿Qué es JPA?

**Spring Boot JPA** es una especificación de Java para administrar datos **relacionales** en aplicaciones Java. Nos permite acceder y conservar datos entre el objeto / clase Java y la base de datos relacional. JPA sigue el **mapeo de relación de objetos** (ORM). Es un conjunto de interfaces. También proporciona una API **EntityManager** en tiempo de ejecución para procesar consultas y transacciones en los objetos en la base de datos. Utiliza un lenguaje de consulta orientado a objetos independiente de la plataforma JPQL (Java Persistent Query Language).



En el contexto de la persistencia, cubre tres áreas:

- La API de persistencia de Java
- Metadatos **relacionales de objeto**
- La propia API, definida en el paquete de **persistencia**

JPA no es un framework. Define un concepto que puede ser implementado por cualquier framework.

## ¿Por qué deberíamos usar JPA?

JPA es más simple, más limpio y menos laborioso que JDBC, SQL y el mapeo escrito a mano. JPA es adecuado para aplicaciones complejas no orientadas al rendimiento. La principal ventaja de JPA sobre JDBC es que, en JPA, los datos están representados por objetos y clases, mientras que en JDBC los datos están representados por tablas y registros. Utiliza POJO para representar datos persistentes que simplifican la programación de la base de datos. Hay algunas otras ventajas de JPA:

- JPA evita escribir DDL en un dialecto SQL específico de la base de datos. En lugar de esto, permite mapear en XML o usar anotaciones de Java.
- JPA nos permite evitar escribir DML en el dialecto SQL específico de la base de datos.
- JPA nos permite guardar y cargar objetos y gráficos Java sin ningún lenguaje DML.
- Cuando necesitamos realizar consultas JPQL, nos permite expresar las consultas en términos de entidades Java en lugar de la tabla y columnas SQL (nativas).

## Funciones de JPA

Hay las siguientes características de JPA:

- Es un repositorio poderoso y una **abstracción de mapeo de objetos** personalizado .
- Es compatible con la **persistencia entre tiendas** . Significa que una entidad se puede almacenar parcialmente en MySQL y Neo4j (Graph Database Management System).





- Genera consultas dinámicamente a partir del nombre de los métodos de consultas.
- Las clases base de dominio proporcionan propiedades básicas.
- Es compatible con la auditoría transparente.
- Posibilidad de integrar código de repositorio personalizado.
- Es fácil de integrar con Spring Framework con el espacio de nombres personalizado.

## Arquitectura JPA

JPA es una fuente para almacenar entidades comerciales como entidades relacionales. Muestra cómo definir un POJO como entidad y cómo gestionar entidades con relación.

La siguiente figura describe la arquitectura de nivel de clase de JPA que describe las clases principales y las interfaces de JPA que se definen en el paquete de **persistencia de javax**. La arquitectura JPA contiene las siguientes unidades:

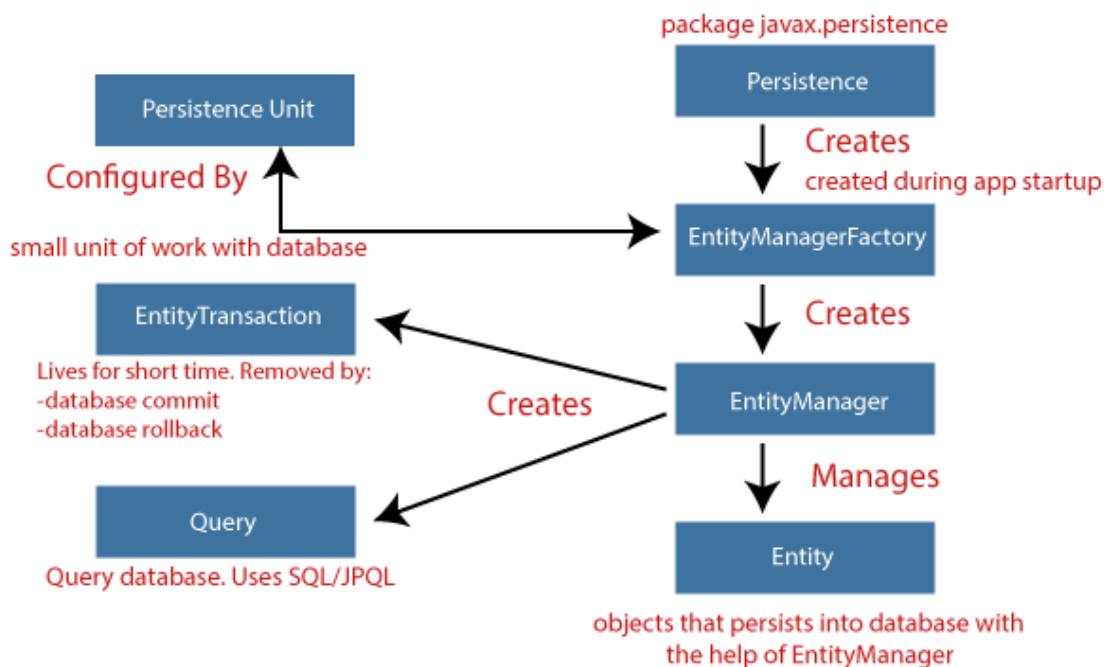
- **Persistence (Persistencia):** es una clase que contiene métodos estáticos para obtener una instancia de EntityManagerFactory.
- **EntityManagerFactory:** es una clase de fábrica de EntityManager. Crea y administra múltiples instancias de EntityManager.
- **EntityManager:** es una interfaz. Controla las operaciones de persistencia en los objetos. Funciona para la instancia de Query.
- **Entity(Entity):** Las entidades son los objetos de persistencia que se almacenan como un registro en la base de datos.
- **Persistence unit (Unidad de persistencia):** define un conjunto de todas las clases de entidad. En una aplicación, las instancias de EntityManager la administran. El conjunto de clases de entidad representa los datos contenidos en un único almacén de datos.
- **EntityTransaction:** tiene una relación de **uno a uno** con la clase EntityManager. Para cada EntityManager, las operaciones se mantienen mediante la clase EntityTransaction.





- **Query (Consulta):** Es una interfaz que es implementada por cada proveedor de JPA para obtener objetos de relación que cumplan con los criterios.

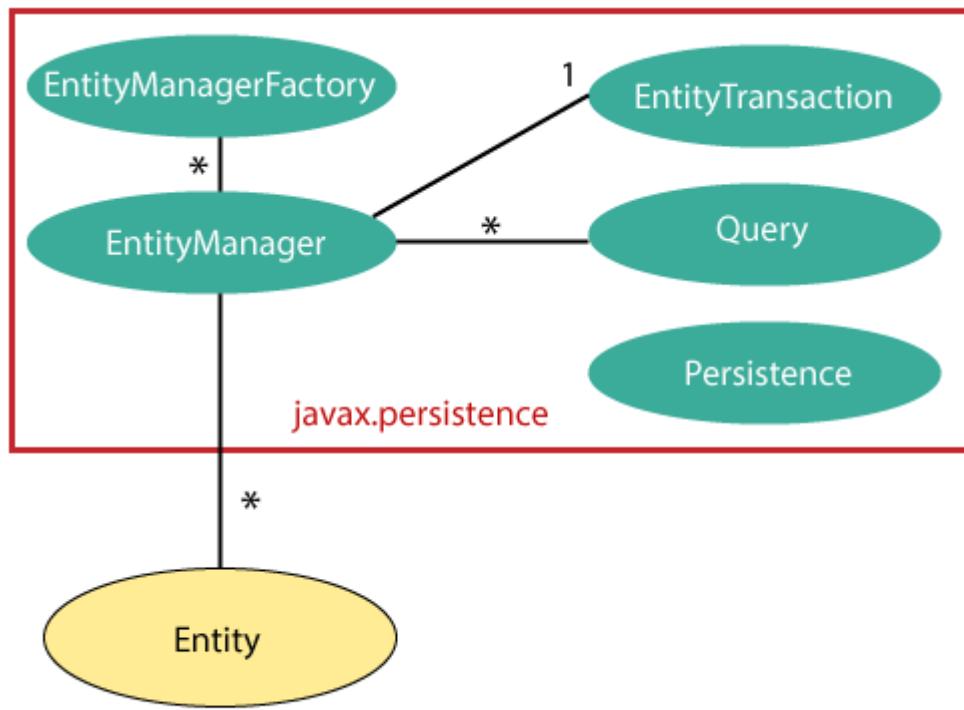
## Architecture of Java Persistence API



## Relaciones de clase JPA

Las clases y las interfaces que hemos discutido anteriormente mantienen una relación. La siguiente figura muestra la relación entre clases e interfaces.





### JPA Class Relationship

- La relación entre EntityManager y EntityTransaction es **uno a uno**. Hay una instancia de EntityTransaction para cada operación de EntityManager.
- La relación entre EntityManagerFactory y EntityManager es de **uno a muchos**. Es una clase de fábrica para la instancia de EntityManager.
- La relación entre EntityManager y Query es de **uno a muchos**. Podemos ejecutar cualquier número de consultas utilizando una instancia de la clase EntityManager.
- La relación entre EntityManager y Entity es de **uno a muchos**. Una instancia de EntityManager puede administrar varias Entidades.

## Implementaciones JPA

JPA es una API de código abierto. Hay varios proveedores de empresas como Eclipse, RedHat, Oracle, etc. que proporcionan nuevos productos agregando el JPA en ellos. Hay algunos marcos de implementación populares de JPA como **Hibernate**, **EclipseLink**, **DataNucleus**, etc. También se conoce como herramienta **Object-Relation Mapping (ORM)**.

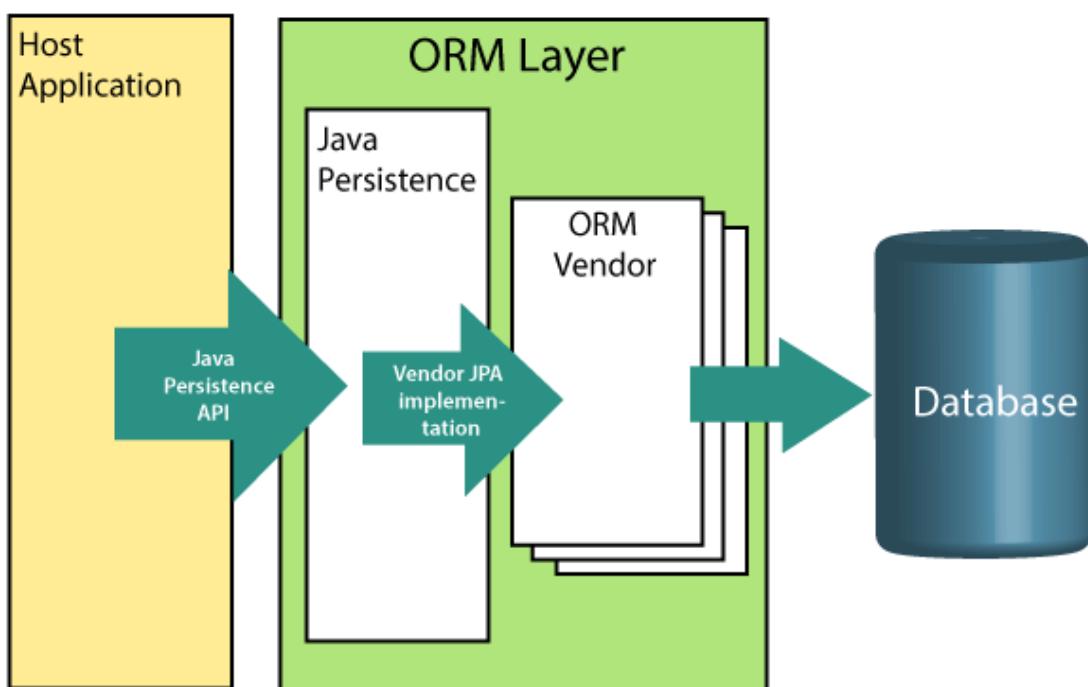




## Mapeo de relación de objeto (ORM)

En ORM, la asignación de objetos Java a tablas de bases de datos y viceversa se denomina **asignación relacional de objetos**. El mapeo ORM funciona como un puente entre una **base de datos relacional** (tablas y registros) y una **aplicación Java** (clases y objetos).

En la siguiente figura, la capa ORM es una capa adaptadora. Adapta el lenguaje de los gráficos de objetos al lenguaje de SQL y tablas de relaciones.



La capa ORM existe entre la aplicación y la base de datos. Convierte las clases y los objetos de Java para que puedan almacenarse y administrarse en una base de datos relacional. De forma predeterminada, el nombre que persiste se convierte en el nombre de la tabla y los campos se convierten en columnas. Una vez que se configura una aplicación, cada fila de la tabla corresponde a un objeto.

## Versiones JPA

Las versiones anteriores de EJB definen la capa de persistencia combinada con la capa de lógica empresarial mediante la interfaz `javax.ejb.EntityBean`. La especificación EJB incluye la definición de JPA.





Al presentar EJB 3.0, la capa de persistencia se separó y se especificó como JPA 1.0 (API de persistencia de Java). Las especificaciones de esta API se publicaron junto con las especificaciones de JAVA EE5 el 11 de mayo de 2006, utilizando JSR 220.

En 2019, JPA pasó a llamarse **Jakarta Persistence**. La última versión de JPA es **2.2**. Admite las siguientes características:

- Java 8, API de datos y tiempo
- Inyección de CDI en AttributeConverters
- Hace anotaciones @Repeatable

## Diferencia entre JPA e Hibernate

**JPA:** JPA es una especificación de Java que se utiliza para acceder, gestionar y conservar datos entre el objeto Java y la base de datos relacional. Es un enfoque estándar para ORM.

**Hibernate:** es una herramienta ORM ligera y de código abierto que se utiliza para almacenar objetos Java en el sistema de base de datos relacional. Es un proveedor de JPA. Sigue un enfoque común proporcionado por JPA.

La siguiente tabla describe las diferencias entre JPA e Hibernate.

<b>JPA</b>	<b>Hibernar</b>
JPA es una <b>especificación de Java</b> para mapear datos de relaciones en una aplicación Java.	Hibernate es un <b>marco ORM</b> que se ocupa de la persistencia de datos.
JPA no proporciona clases de implementación.	Proporciona clases de implementación.
Utiliza un lenguaje de consulta independiente de la plataforma llamado <b>JPQL</b> (Java Persistence Query Language).	Utiliza su propio lenguaje de consulta llamado <b>HQL</b> (Hibernate Query Language).





Está definido en el paquete <b>javax.persistence</b> .	Está definido en el paquete <b>org.hibernate</b> .
Está implementado en varias herramientas ORM como <b>Hibernate, EclipseLink</b> , etc.	Hibernate es el <b>proveedor</b> de JPA.
JPA usa <b>EntityManager</b> para manejar la persistencia de datos.	En Hibernate usa <b>Session</b> para manejar la persistencia de datos.

## Spring Boot Starter Data JPA

Spring Boot proporciona la dependencia de arranque **spring-boot-starter-data-jpa** para conectar la aplicación Spring Boot con la base de datos relacional de manera eficiente. Spring-boot-starter-data-jpa usa internamente la dependencia **spring-boot-jpa**.

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>**

## Ejemplo de Spring Boot JPA

Creemos una aplicación Spring Boot que use JPA para conectarse a la base de datos. En el siguiente ejemplo, usamos la base de datos en memoria **Apache Derby**.

**Apache Derby:** es una base de datos relacional **incrustada de código abierto** implementada completamente en Java. Está disponible bajo la licencia Apache 2.0. Apache Derby presenta las siguientes ventajas:

- Es fácil de instalar, implementar y usar.
- Se basa en los estándares Java, JDBC y SQL.
- Proporciona un controlador JDBC integrado que nos permite integrar Derby en cualquier solución basada en Java.



- También es compatible con el modo cliente / servidor con el controlador JDBC de Derby Network Client y Derby Network Server.

Spring Boot puede configurar automáticamente una base de datos incrustada como **H2**, **HSQL** y **Derbydatabases**. No es necesario que proporcionemos ninguna URL de conexión. Solo necesitamos incluir una dependencia de compilación en la base de datos incrustada que queremos usar.

En Spring Boot, podemos integrar fácilmente la base de datos Apache Derby simplemente agregando la dependencia **Derby** en el archivo pom.xml.

1. **<dependency>**
2. **<groupId>org.apache.derby</groupId>**
3. **<artifactId>derby</artifactId>**
4. **<scope>runtime</scope>**
5. **</dependency>**

**Paso 1:** Abra Spring Initializr <https://start.spring.io/> .

**Paso 2:** seleccione la última versión de Spring Boot **2.3.0 (SNAPSHOT)**

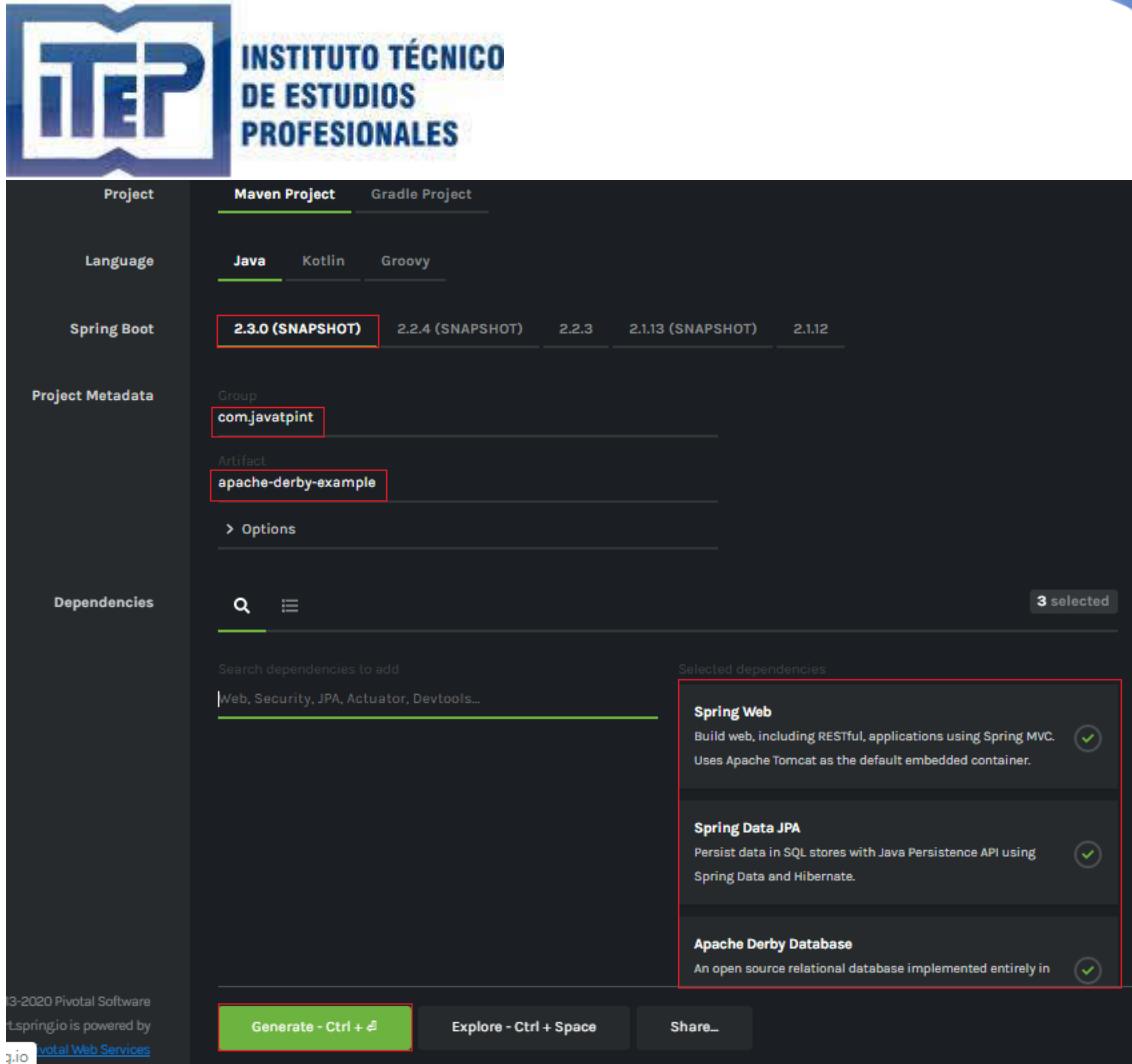
**Paso 3:** proporcione el nombre del **grupo** . Hemos proporcionado **com.javatpoint**.

**Paso 4:** proporcione la identificación del **artefacto** . Hemos proporcionado **apache-derby-example** .

**Paso 5:** agregue las dependencias: **Spring Web**, **Spring Data JPA** y **Apache Derby Database** .

**Paso 6:** Haga clic en el botón **Generar** . Cuando hacemos clic en el botón Generar, envuelve el proyecto en un archivo Jar y lo descarga al sistema local.





**Paso 7:** extraiga el archivo Jar y péguelo en el espacio de trabajo de STS.

**Paso 8: Importe** la carpeta del proyecto a STS.

Archivo -> Importar -> Proyectos Maven existentes -> Examinar -> Seleccione la carpeta apache-derby-example -> Finalizar

La importación lleva algún tiempo.

**Paso 9:** Cree un paquete con el nombre **com.javatpoint.model** en la carpeta **src/main/java**.

**Paso 10:** Cree una clase con el nombre **UserRecord** en el paquete **com.javatpoint.model** y haga lo siguiente:

- Defina tres variables **id, nombre y correo electrónico** .
  - Genere Getters y Setter.
- Haga clic derecho en el archivo -> Fuente -> Generar Getters y Setters



- Defina un constructor predeterminado.
- Marque la clase como una **entidad** utilizando la anotación **@Entity**.
- Marque **Id** como la clave principal utilizando la anotación **@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 constructor
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. }
```

**Paso 11:** Cree un paquete con el nombre **com.javatpoint.controller** en la carpeta **src / main / java**.

**Paso 12:** Cree una clase de controlador con el nombre **UserController** en el paquete **com.javatpoint.controller** y haga lo siguiente:

- Marque la clase como controlador mediante la anotación **@RestController**.
- Crea un cableado automático con la clase **UserService** mediante la anotación **@Autowired**.
- Hemos definido dos asignaciones, una para **obtener todos los usuarios** y la otra para **agregar usuarios**.

### 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. }
```

**Paso 13:** Cree un paquete con el nombre **com.javatpoint.service** en la carpeta **src / main / java**.

**Paso 14:** Cree una clase de servicio con el nombre **UserController** en el paquete **com.javatpoint.service** y haga lo siguiente:

- Marque la clase como servicio utilizando la anotación **@Service**.
- Autowired el **UserRepository**
- Defina un método **getAllUsers ()** que devuelva una lista de
- Defina otro nombre de método **addUser ()** que **guarde** el registro del usuario.





## 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. }
```

**Paso 15:** Cree un paquete con el nombre **com.javatpoint.repository** en la carpeta **src / main / java**.

**Paso 16:** Cree una interfaz de repositorio con el nombre  **UserRepository** en el paquete **com.javatpoint.repository** y amplíe **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, S
tring>
5. {
6. }

```

**Paso 17:** Ahora, abra el archivo **ApacheDerbyExampleApplication.java**. Se crea de forma predeterminada cuando configuramos una aplicación.

### 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. }

```

Ahora, hemos configurado todas las clases y paquetes necesarios de acuerdo con los requisitos. Tenga en cuenta que no hemos proporcionado ninguna **URL de conexión** para la base de datos. Después de completar todos los pasos anteriores, el directorio del proyecto tiene el siguiente aspecto:





```

M S apache-derby-example [boot]
  ▾ src/main/java
    ▾ com.javatpoint
      ▾ ApacheDerbyExampleApplication.java
    ▾ com.javatpoint.controller
      ▾ UserController.java
    ▾ com.javatpoint.model
      ▾ UserRecord.java
    ▾ com.javatpoint.repository
      ▾ UserRepository.java
    ▾ com.javatpoint.service
      ▾ UserService.java
  ▾ src/main/resources
  ▾ src/test/java
  ▾ JRE System Library [JavaSE-1.8]
  ▾ Maven Dependencies
  ▾ src
    ▾ target
    HELP.md
    mvnw
    mvnw.cmd
  pom.xml

```

Ejecutemos la aplicación.

**Paso 18:** Abra el archivo **ApacheDerbyExampleApplication.java** y ejecútelo como aplicación Java.

**Paso 19:** Abra el navegador e invoque la URL `http://localhost:8080/`. Devuelve una lista vacía porque no hemos agregado ningún usuario en la lista.

Para agregar un usuario a la base de datos, enviaremos una solicitud **POST** utilizando **Postman**.

**Paso 20:** Abra **postman** y haga lo siguiente:

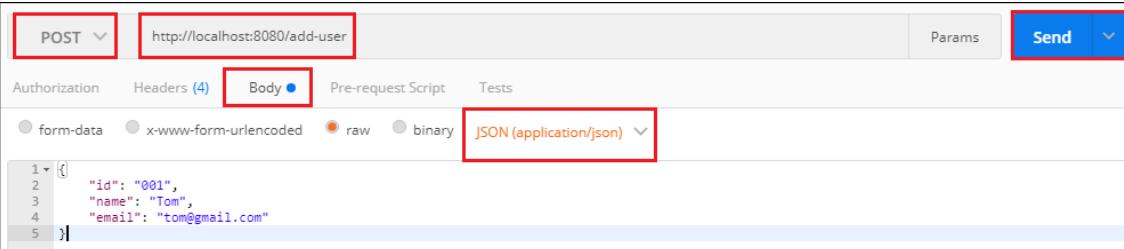
- Seleccione el **POST**
- Invoque la URL `http://localhost:8080/add-user`.
- Haga clic en el **body**
- Seleccione Content-Type como **JSON** (aplicación / json).
- Inserte los datos que deseé insertar en la base de datos. Hemos insertado los siguientes datos:



```

1. {
2.   "id": "001",
3.   "name": "Tom",
4.   "email": "tom@gmail.com"
5. }
```

- Haga clic en el botón **Enviar**.



POST  Send

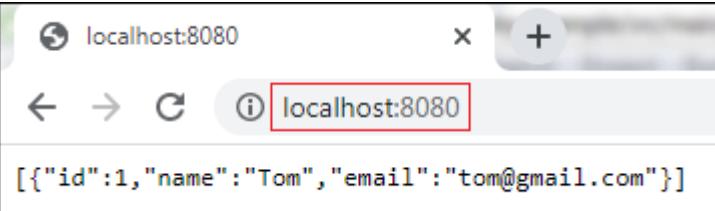
Body (JSON application/json)

```

1 [{}]
2   "id": "001",
3   "name": "Tom",
4   "email": "tom@gmail.com"
5 ]
```

Cuando hacemos clic en el botón Enviar, muestra **Estado: 200 OK**. Significa que la solicitud se ha ejecutado con éxito.

**Paso 21:** Abra el navegador e invoque la URL `http://localhost:8080`. Devuelve el usuario que hemos insertado en la base de datos.



localhost:8080

localhost:8080

```
[{"id": 1, "name": "Tom", "email": "tom@gmail.com"}]
```

[Descargar el proyecto de ejemplo de Apache derby](#)

## Spring Boot JDBC

**Spring Boot JDBC** proporciona un iniciador y bibliotecas para conectar una aplicación con JDBC.





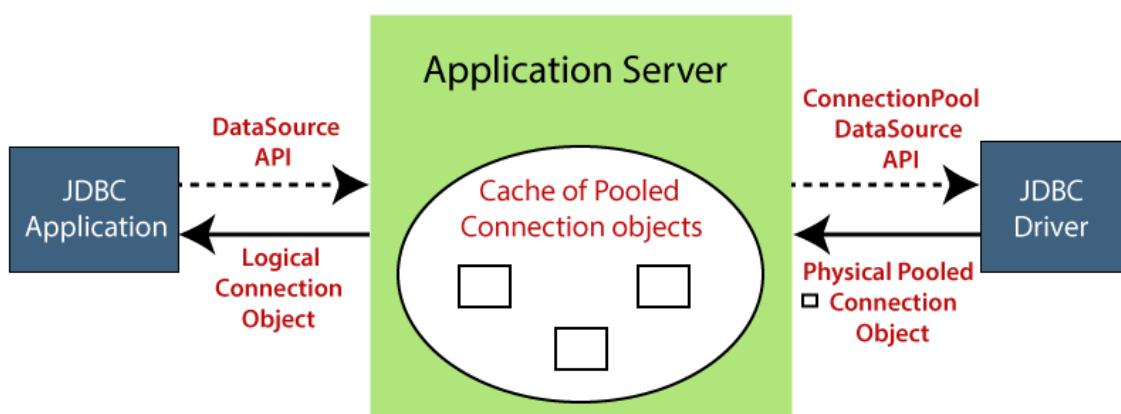
En Spring Boot JDBC, los beans relacionados con la base de datos como **DataSource**, **JdbcTemplate** y **NamedParameterJdbcTemplate** se configuran automáticamente y se crean durante el inicio. Podemos conectar automáticamente estas clases si queremos usarlo. Por ejemplo:

1. @Autowired
2. JdbcTemplate jdbcTemplate;
3. @Autowired
4. **private** NamedParameterJdbcTemplate jdbcTemplate;

En el archivo **application.properties**, configuramos **DataSource** y la **agrupación de conexiones**. [Spring\\_Boot](#) elige la agrupación de **tomcat** de forma predeterminada.

## Agrupación de conexiones JDBC

**La agrupación de conexiones JDBC** es un mecanismo que gestiona **varias** solicitudes de conexión a la base de datos. En otras palabras, facilita la reutilización de conexiones, una memoria caché de conexiones de bases de datos, denominada **agrupación de conexiones**. Un módulo de agrupación de conexiones lo mantiene como una capa sobre cualquier producto de controlador JDBC estándar.

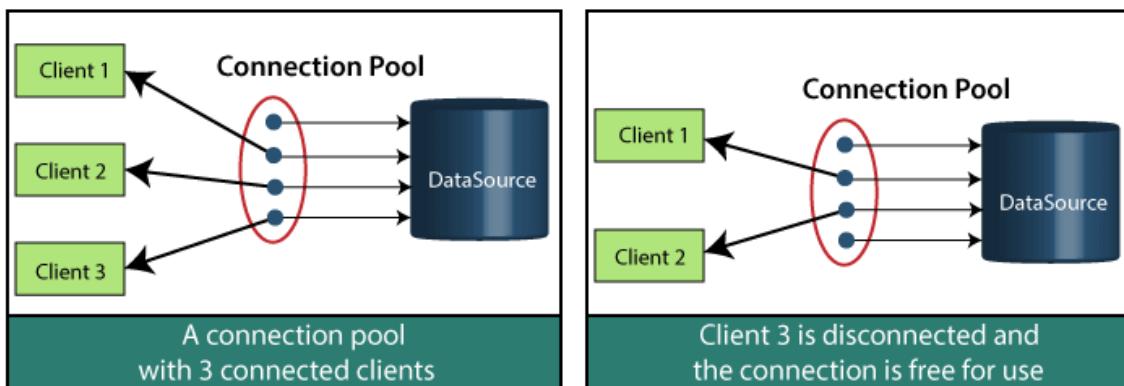


Aumenta la velocidad de acceso a los datos y reduce el número de conexiones a la base de datos para una aplicación. También mejora el rendimiento de una aplicación. El grupo de conexiones realiza las siguientes tareas:





- Administrar la conexión disponible
- Asignar nueva conexión
- Conexión cercana



En la figura anterior, hay **clientes**, un **grupo de conexiones** (que tiene cuatro conexiones disponibles) y un **DataSource**.

En la primera figura, hay tres clientes conectados con diferentes conexiones y hay una conexión disponible. En la segunda figura, el Cliente 3 se ha desconectado y esa conexión está disponible.

Cuando un cliente completa su trabajo, libera la conexión y esa conexión está disponible para otros clientes.

## HikariCP

El grupo de conexiones predeterminado en Spring Boot 2 es **HikariCP**. Proporciona funciones listas para la empresa y un mejor rendimiento. HikariCP es una implementación de JDBC DataSource que proporciona un mecanismo de agrupación de conexiones.

- Si el HikariCP está presente en la ruta de clases, Spring Boot lo configura automáticamente.
- Si el HikariCP no se encuentra en la ruta de clases, Spring Boot busca el grupo de **conexiones Tomcat JDBC**. Si está en la ruta de clase Spring Boot, recójalo.
- Si las dos opciones anteriores no están disponibles, Spring Boot elige **Apache Commons DBCP2** como el grupo de conexiones JDBC.





También podemos configurar un grupo de conexiones manualmente, si no queremos usar el grupo de conexiones predeterminado. Supongamos que queremos usar el grupo de conexiones Tomcat JDBC en lugar de HikariCP. Se excluyeron dependencia **HikariCP** y añadir la dependencia **Tomcat-JDBC** en el archivo pom.xml, como se muestra a continuación.

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>**

El enfoque anterior nos permite usar el grupo de conexiones Tomcat sin tener que escribir una clase **@Configuration** y definir programáticamente un bean **DataSource**.

Por otro lado, también podemos omitir el algoritmo de escaneo del grupo de conexiones que usa Spring Boot. Podemos especificar explícitamente una **fuente**





de datos de agrupación de conexiones agregando la propiedad **spring.datasource.type** en el archivo application.properties.

1. **Spring.datasource.type=org.apache.tomcat.jdbc.pool.DataSource**

Hemos configurado el grupo de conexiones de Tomcat. Ahora, agregaremos algunas propiedades en **application.properties** que optimizan su rendimiento y se adaptan a algunos requisitos específicos.

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**

Si queremos conectarnos a la base de datos MySQL , necesitamos incluir el controlador JDBC en la ruta de clase de la aplicación:

1. **<!-- MySQL JDBC driver -->**
2. **<dependency>**
3. **<groupId>mysql</groupId>**
4. **<artifactId>mysql-connector-java</artifactId>**
5. **</dependency>**

Después de eso, defina las propiedades del **datasoure** en el archivo **application.properties** .

Utilice las siguientes propiedades si está utilizando una **base de datos MySQL** :



1. `spring.datasource.url=jdbc:mysql://192.168.1.4:3306/test`
2. `spring.datasource.username=javatpoint`
3. `spring.datasource.password=password`

Utilice las siguientes propiedades si está utilizando una base de datos **Oracle** :

1. `spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl`
2. `spring.datasource.username=system`
3. `spring.datasource.password=Password123`

**Nota:** Spring Boot 2 usa HikariCP como grupo de conexión de base de datos, de forma predeterminada. Si el HikariCP no está presente en la ruta de clase, Spring Boot elige la agrupación de tomcat de forma predeterminada.

## ¿Por qué deberíamos usar Spring Boot JDBC?

La funcionalidad de Spring JDBC y Spring Boot JDBC es la misma, excepto las implementaciones. Existen las siguientes ventajas de Spring Boot JDBC sobre Spring JDBC:

Spring Boot JDBC	Primavera JDBC
Solo se requiere una dependencia <b>spring-boot-starter-jdbc</b> .	En Spring JDBC, se deben configurar múltiples dependencias como <b>spring-jdbc</b> y <b>spring-context</b> .
Configura automáticamente el bean de origen de datos, si no se mantiene explícitamente. Si no queremos usar el bean, podemos establecer una propiedad <b>spring.datasource.initialize</b> en <b>false</b> .	En Spring JDBC, es necesario crear un bean de base de datos utilizando <b>XML</b> o <b>javaconfig</b> .



No necesitamos registrar los beans de plantilla porque Spring Boot los registra automáticamente.	Los beans de plantilla como <b>PlatformTransactionManager</b> , <b>JdbcTemplate</b> , <b>NamedParameterJdbcTemplate</b> deben estar registrados.
Todos los scripts de inicialización de la base de datos almacenados en el archivo .sql se ejecutan automáticamente.	Si se crea algún script de inicialización de la base de datos, como la eliminación o la creación de tablas, en un archivo SQL, esta información debe proporcionarse explícitamente en la configuración.

## JDBC frente a hibernación

JDBC	Hibernar
JDBC es una <b>tecnología</b> .	Hibernate es un marco <b>ORM</b> .
En JDBC, el usuario es responsable de crear y cerrar las conexiones.	En Hibernate, el sistema de tiempo de ejecución se encarga de crear y cerrar las conexiones.
No es compatible con la carga diferida.	Es compatible con la carga diferida que ofrece un mejor rendimiento.
No admite asociaciones (la conexión entre dos clases separadas).	Apoya asociaciones.

En la siguiente sección, aprenderemos la conectividad de MySQL en una aplicación Spring Boot.

## Ejemplo de Spring Boot JDBC

Spring Boot proporciona un iniciador y bibliotecas para conectarse a nuestra aplicación con JDBC. Aquí, estamos creando una aplicación que se conecta con la





base de datos Mysql. Incluye los siguientes pasos para crear y configurar JDBC con Spring Boot

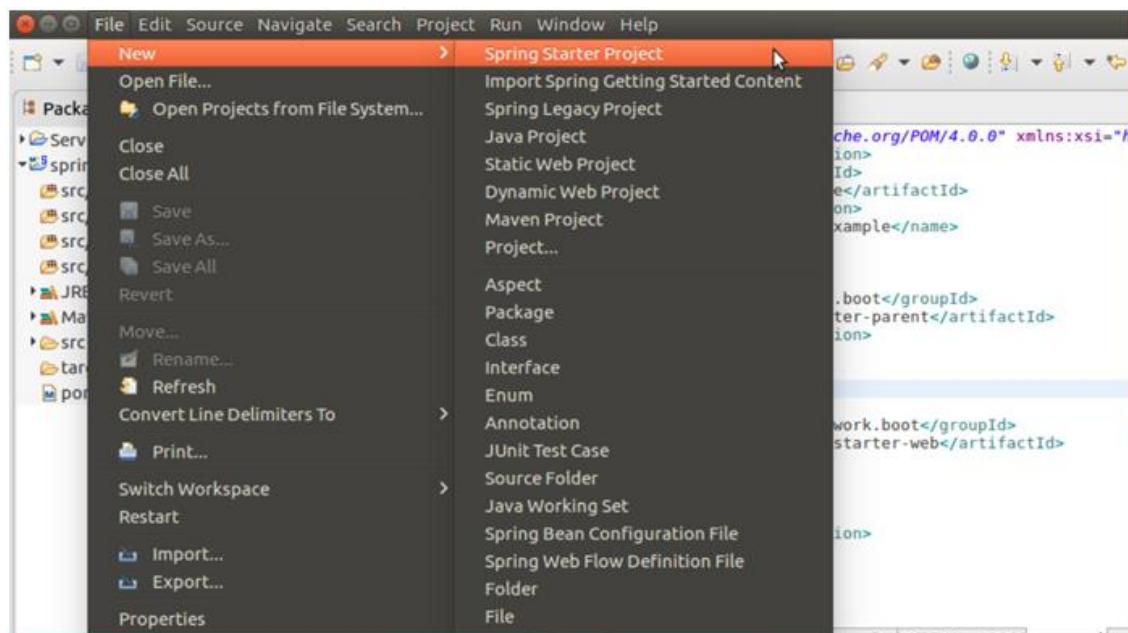
Crea una base de datos

1. create database springbootdb

Crea una tabla en MYSQL

1. create table user(id **int** UNSIGNED primary key not **null** auto\_increment, name varchar(**100**), email varchar(**100**));

La creación de un Spring Boot Project



Proporcionar el nombre del proyecto y otra información relacionada con el proyecto.





New Spring Starter Project

This screenshot shows the "New Spring Starter Project" dialog. The "Name" field is set to "spring-boot-JDBC". Other fields include "Type: Maven", "Java Version: 1.8", "Group: com.javatpoint", "Artifact: spring-boot-JDBC", "Version: 0.0.1-SNAPSHOT", "Description: A new project of Spring Boot JDBC", and "Package: com.javatpoint". The "Working sets" section has a checked checkbox for "Add project to working sets". Buttons at the bottom include "?", "< Back", "Next >", "Cancel", and "Finish".

## Proporcionar dependencias

New Spring Starter Project

This screenshot shows the "Dependencies" section of the "New Spring Starter Project" dialog. The "Spring Boot Version" is set to "2.0.0 (SNAPSHOT)". Under the "Web" category, the "JDBC" and "MySQL" checkboxes are selected. Other categories like "Core", "SQL", and "Web" have their respective checkboxes for "Validation", "Reactive Web", "Websocket", "Rest Repositories", and "Mobile" unselected. Buttons at the bottom include "?", "< Back", "Next >", "Cancel", and "Finish".

Después de terminar, cree los siguientes archivos en su proyecto.





Configure la base de datos en el archivo application.properties.

```
// 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. }

Creando un controlador para manejar solicitudes HTTP.

```
// 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;





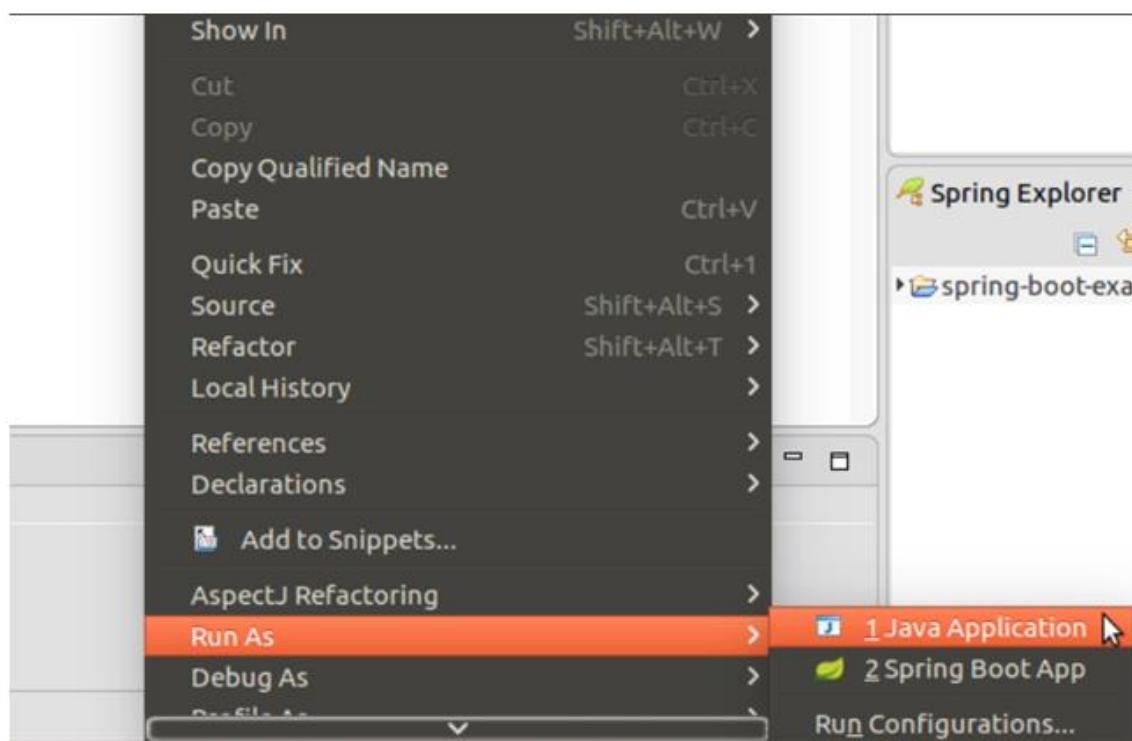
```

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@ja
vatpoint.com')");
13.        return "data inserted Successfully";
14.    }
15.}

```

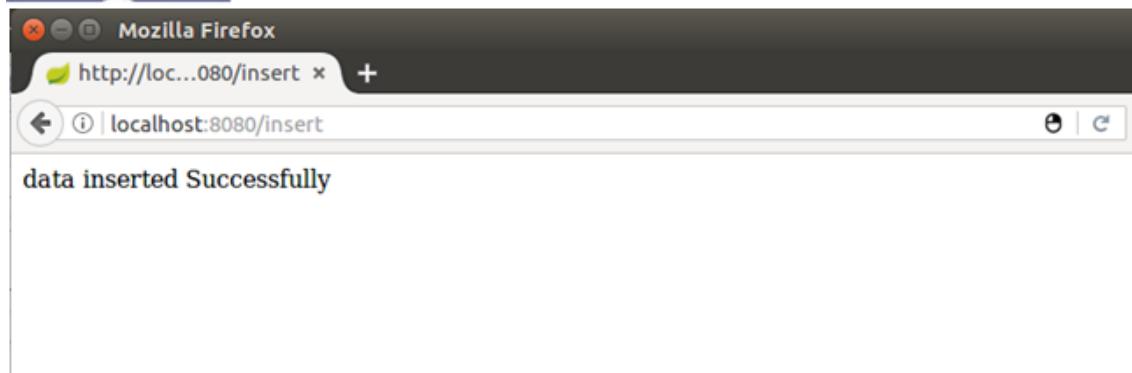
## Ejecuta la aplicación

Ejecute el archivo **SpringBootJdbcApplication.java** como aplicación Java .



Ahora, abra el navegador y siga la siguiente URL.





Dice que los datos se han insertado correctamente. confirmemos comprobando la tabla mysql.

```
mysql> use springbootdb;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from user;
+----+-----+-----+
| id | name | email |
+----+-----+-----+
| 3 | javatpoint | java@javatpoint.com |
+----+-----+
1 row in set (0.00 sec)

mysql>
```

Bueno, nuestra aplicación funciona bien. Ahora, también podemos realizar otras operaciones de base de datos.

## Base de datos Spring Boot H2

### ¿Qué es la base de datos en memoria?

La base de datos en memoria se basa en la memoria del sistema en oposición al espacio en disco para el almacenamiento de datos. Porque el acceso a la memoria es más rápido que el acceso al disco. Usamos la base de datos en memoria cuando no necesitamos conservar los datos. La base de datos en memoria es una base de datos incorporada. Las bases de datos en memoria son volátiles, por defecto, y todos los datos almacenados se pierden cuando reiniciamos la aplicación.

Las bases de datos en memoria ampliamente utilizados son **H2**, **HSQldb** (HyperSQL base de datos), y **Apache Derby**. Crea la configuración automáticamente.



## Persistencia frente a base de datos en memoria

La base de datos persistente conserva los datos en la memoria física. Los datos estarán disponibles incluso si se rebota el servidor de la base de datos. Algunas bases de datos de persistencia populares son **Oracle** , **MYSQL** , **Postgres** , etc.

En el caso de la **base de datos en memoria**, **los** datos se almacenan en la **memoria del sistema** . Perdió los datos cuando se cerró el programa. Es útil para **POC** (Prueba de conceptos), no para una aplicación de producción. La base de datos en memoria más utilizada es **H2**.

### ¿Qué es la base de datos H2?

**H2** es una base de datos **integrada, de código abierto y en memoria** . Es un sistema de gestión de bases de datos relacionales escrito en Java. Es una aplicación **cliente / servidor** . Generalmente se usa en **pruebas unitarias** . Almacena datos en la memoria, no conserva los datos en el disco.

#### Ventajas

- Configuración cero
- Es fácil de usar.
- Es ligero y rápido.
- Proporciona una configuración sencilla para cambiar entre una base de datos real y una base de datos en memoria.
- Es compatible con la API estándar de SQL y JDBC.
- Proporciona una consola web para mantener en la base de datos.

### Configurar la base de datos H2

Si queremos usar la base de datos H2 en una aplicación, debemos agregar la siguiente dependencia en el archivo pom.xml:

1. **<dependency>**
2. **<groupId>com.h2database</groupId>**
3. **<artifactId>h2</artifactId>**
4. **<scope>runtime</scope>**
5. **</dependency>**





Después de agregar la dependencia, necesitamos configurar **la URL de la fuente de datos, el nombre de la clase del controlador, el nombre de usuario y la contraseña** de la base de datos H2. Spring Boot proporciona una manera fácil de configurar estas propiedades en el archivo **application.properties**.

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

En la propiedad **spring.datasource.url**, **mem** es el nombre de una base de datos en memoria y **testdb** es el nombre del esquema que proporciona H2, de forma predeterminada. También podemos definir nuestro propio esquema y base de datos. El nombre de usuario predeterminado es **sa** y la contraseña en blanco denota una contraseña **vacía**. Si queremos cambiar el nombre de usuario y la contraseña, podemos anular estos valores.

## Conservar los datos en la base de datos H2

Si queremos conservar los datos en la base de datos H2, debemos almacenar los datos en un archivo. Para lograr lo mismo, necesitamos cambiar la propiedad de la URL de la fuente de datos.

1. #persist the data
2. spring.datasource.url=jdbc:h2:file:/data/sampledata
3. spring.datasource.url=jdbc:h2:C:/data/sampledata

En la propiedad anterior, sampledata es un nombre de archivo.





## Crear esquema y completar datos

Podemos definir el esquema creando un archivo **SQL** en la carpeta de recursos (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. );

Podemos completar datos en la tabla creando un archivo **SQL** en la carpeta de **recursos** (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 recoge automáticamente el archivo **data.sql** y lo ejecuta en la base de datos H2 durante el inicio de la aplicación.

## Consola H2

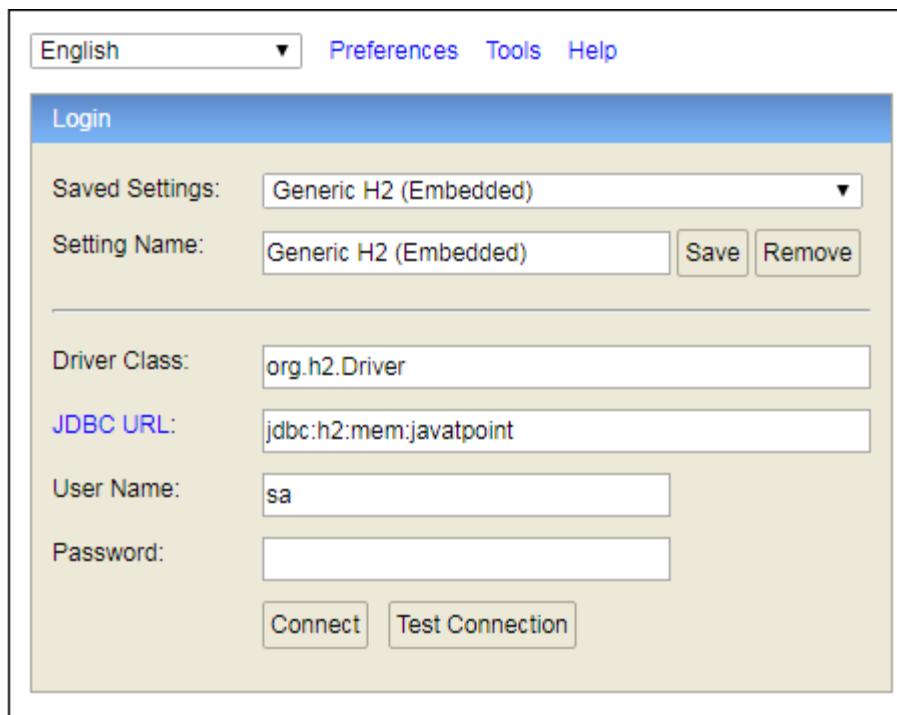
De forma predeterminada, la vista de la consola de la base de datos H2 está deshabilitada. Antes de acceder a la base de datos H2, debemos habilitarla mediante la siguiente propiedad.

1. #enabling the H2 console



## 2. spring.h2.console.enabled=true

Una vez habilitada la consola H2, ahora podemos acceder a la consola H2 en el navegador invocando la URL `http://localhost: 8080 / h2-console`. La siguiente figura muestra la vista de la consola de la base de datos H2.



En la captura de pantalla anterior, hemos definido nuestra propia base de datos llamada **javatpoint**.

## Ejemplo de Spring Boot H2

Configuremos una aplicación Spring Boot con la base de datos H2.

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

**Paso 2:** seleccione Spring Boot versión **2.3.0.M1**.

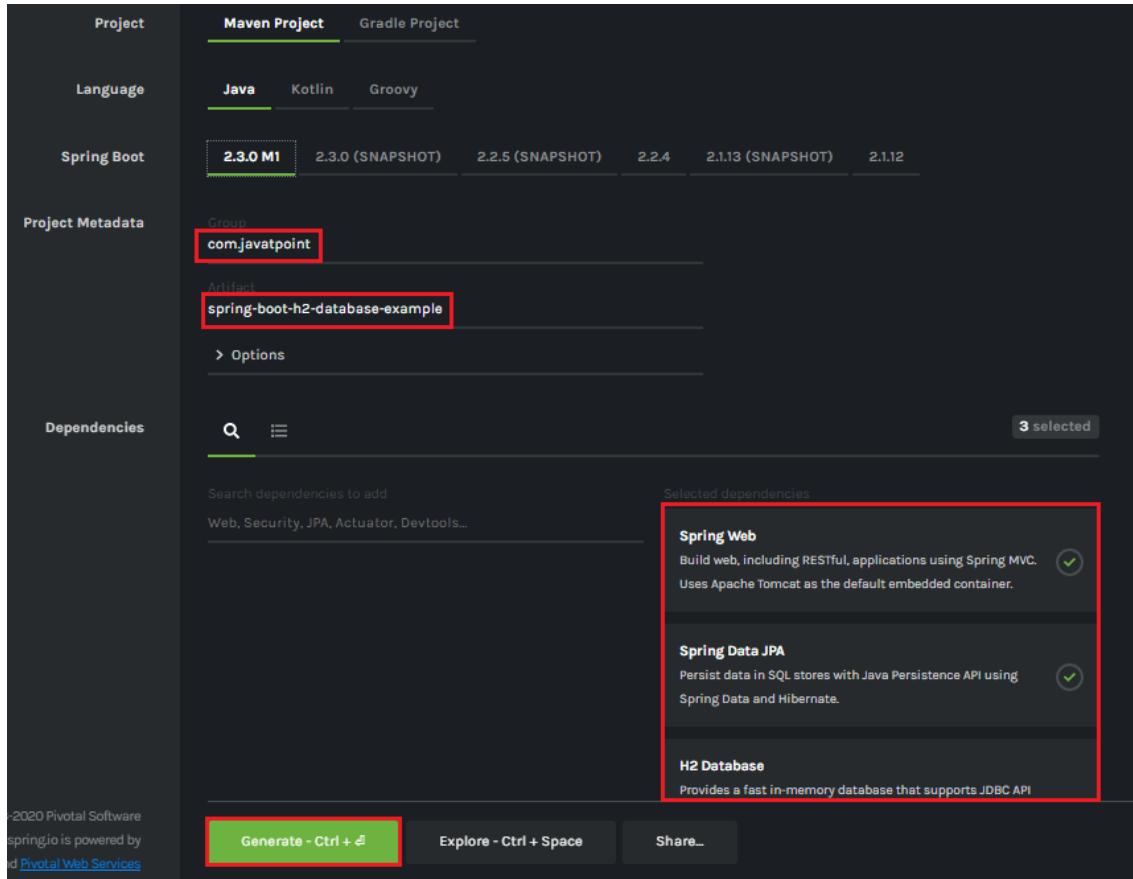
**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado **com.javatpoint**.

**Paso 3:** proporcione el Id. Del **artefacto** . Hemos proporcionado **spring-boot-h2-database-example**.



**Paso 5:** Añadir las dependencias **Spring Web**, **Spring Data JPA**, y **H2 Database**.

**Paso 6:** Haga clic en el botón **Generar**. Cuando hacemos clic en el botón Generar, envuelve el proyecto en un archivo **Jar** y lo descarga al sistema local.



**Paso 7:** extraiga el archivo Jar y péguelo en el espacio de trabajo de STS.

**Paso 8: Importe** la carpeta del proyecto a STS.

Archivo -> Importar -> Proyectos existentes de Maven -> Examinar -> Seleccione la carpeta `spring-boot-h2-database-example` -> Finalizar

La importación lleva algún tiempo.

**Paso 9:** Cree un paquete con el nombre **com.javatpoint.model** en la carpeta **src / main / java**.





**Paso 10:** Cree una clase de modelo en el paquete **com.javatpoint.model**. Hemos creado una clase modelo con el nombre **Student**. En la clase Books, hemos hecho lo siguiente:

- Defina cuatro variables **id, age, name** y
- Genere Getters y Setters.  
Haga clic derecho en el archivo -> Fuente -> Generar Getters y Setters.
- Marque la clase como **Entidad** usando la anotación **@Entity**.
- Marque la clase como Nombre de **tabla** usando la anotación **@Table**.
- Defina cada variable como **Columna** utilizando la anotación **@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.}

**Paso 11:** Cree un paquete con el nombre **com.javatpoint.controller** en la carpeta **src / main / java**.

**Paso 12:** Cree una clase de controlador en el paquete **com.javatpoint.controller**. Hemos creado una clase de controlador con el nombre **StudentController**. En la clase StudentController, hemos hecho lo siguiente:

- Marque la clase como **RestController** usando la anotación **@RestController**.
- Conecte automáticamente la clase **StudentService** mediante la anotación **@Autowired**.
- Defina los siguientes métodos:
  - **getAllStudent ()**: devuelve una lista de todos los estudiantes.
  - **getStudent ()**: Devuelve un detalle de estudiante que hemos especificado en la variable de ruta. Hemos pasado id como argumento usando la anotación **@PathVariable**. La anotación indica que un parámetro de método debe estar vinculado a una variable de plantilla de URI.
  - **deleteStudent ()**: Elimina un alumno específico que hemos especificado en la variable de ruta.
  - **saveStudent ()**: Guarda los detalles del estudiante. La anotación **@RequestBody** indica que un parámetro de método debe estar vinculado al cuerpo de la solicitud web.

### **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 da
tabase
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. }
```

**Paso 13:** Cree un paquete con el nombre **com.javatpoint.service** en la carpeta **src / main / java**.

**Paso 14:** Cree una clase de **servicio**. Hemos creado una clase de servicio con el nombre **StudentService** en el paquete **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. }
```

**Paso 15:** Cree un paquete con el nombre **com.javatpoint.repository** en la carpeta **src / main / java**.

**Paso 16:** Cree una interfaz de **repositorio**. Hemos creado una interfaz de repositorio con el nombre **StudentRepository** en el paquete **com.javatpoint.repository**. Extiende la interfaz de **Crud Repository**.

### **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. }

Ahora configuraremos la **URL de** la fuente de datos , **el nombre de la clase del controlador, el nombre de usuario y la contraseña** en el archivo **application.properties** .

**Paso 17:** Abra el archivo **application.properties** y configure las siguientes propiedades.

#### **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**

**Nota: No olvide habilitar la consola H2.**

Después de crear todas las clases y paquetes, el directorio del proyecto tiene el siguiente aspecto.





```

M:~$ spring-boot-h2-database-example [boot]
  ▾ src/main/java
    ▾ com.javatpoint
      ▾ SpringBootH2DatabaseExampleApplication.java
    ▾ com.javatpoint.controller
      ▾ StudentController.java
    ▾ com.javatpoint.model
      ▾ Student.java
    ▾ com.javatpoint.repository
      ▾ StudentRepository.java
    ▾ com.javatpoint.service
      ▾ StudentService.java
  ▾ src/main/resources
    static
    templates
    application.properties
  ▾ src/test/java
  ▾ JRE System Library [JavaSE-1.8]
  ▾ Maven Dependencies
  ▾ src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml

```

Ahora ejecutaremos la aplicación.

**Paso 18:** Abra el archivo **SpringBootH2DatabaseExampleApplication.java** y ejecútelo como una aplicación Java.

### 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. }
```

En el siguiente paso, vamos a utilizar el cliente rest postman para enviar solicitud **POST** y **GET** . Si Postman no está instalado en su sistema, siga los pasos a continuación:

- Descargue Postman de <https://www.getpostman.com/downloads/> o agregue la extensión de Google Chrome en el navegador <https://bit.ly/1HCOcWf> .
- Inicie Postman y **regístrate** . Crear un nombre de usuario. Hemos creado un usuario con el nombre **javatpoint** y **hemos** hecho clic en **Enviar**.

**Paso 19:** Abra postman y haga lo siguiente:

- Seleccione el **POST**
- Invoque la URL `http://localhost:8080/student`.
- Seleccione el **body**
- Seleccione el **JSON** como tipo de contenido (**aplicación / json**).
- Inserte los datos. Hemos insertado los siguientes datos en el cuerpo:

```
{
  "id": "001",
  "age": "23",
  "name": "Amit",
  "email": "amit@yahoo.co.in"
}
```

- Haga clic en **Enviar**

Cuando la solicitud se ejecuta con éxito, muestra el **Estado: 200 OK** . Significa que el registro se ha insertado correctamente en la base de datos.





Del mismo modo, hemos insertado los siguientes datos.

```
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.}
```

Accedamos a la consola H2 para ver los datos.

**Paso 20:** Abra el navegador e invoque la URL `http://localhost:8080/h2-console`. Haga clic en el botón **Conectar**, como se muestra a continuación.



INSTITUTO TÉCNICO  
DE ESTUDIOS  
PROFESIONALES

H2 Console

localhost:8080/h2-console/login.jsp?jsessionid=83c78

English Preferences Tools Help

### Login

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:javatpoint

User Name: sa

Password:

**Connect** Test Connection

Después de hacer clic en el botón **Conectar**, vemos la tabla de **Students** en la base de datos, como se muestra a continuación.

jdbc:h2:mem:javatpoint

- STUDENT
  - + ID
  - + AGE
  - + EMAIL
  - + NAME
  - + Indexes
- + INFORMATION\_SCHEMA
- + Users
- (i) H2 1.4.200 (2019-10-14)

**Paso 21:** Haga clic en la tabla de **Students** y luego haga clic en el botón **Ejecutar**. La tabla muestra los datos que hemos insertado en el cuerpo.





```
Run Run Selected Auto complete Clear SQL statement:  
SELECT * FROM STUDENT ;  
  
SELECT * FROM STUDENT ;  


| ID | AGE | EMAIL              | NAME    |
|----|-----|--------------------|---------|
| 1  | 23  | amit@gmail.com     | Amit    |
| 2  | 24  | vadik@gmail.com    | Vadik   |
| 3  | 21  | prateek@gmail.com  | Prateek |
| 4  | 25  | harsh@yahoo.com    | Harsh   |
| 5  | 24  | swarit@yahoo.co.in | Swarit  |



(5 rows, 483 ms)



Edit

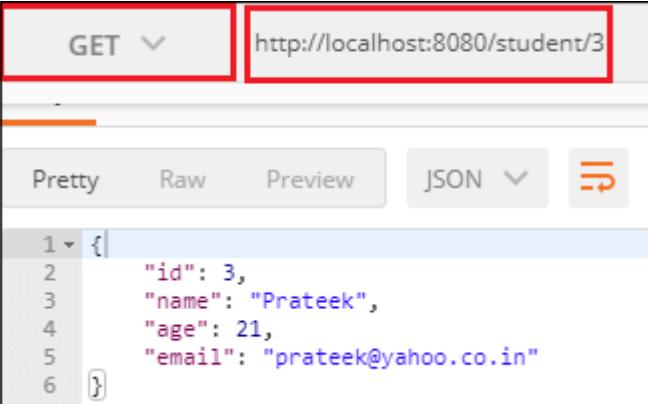

```

**Paso 22:** Abra postman y envíe una solicitud **GET** . Devuelve los datos que hemos insertado en la base de datos.



```
[]
{
    "id": 1,
    "name": "Amit",
    "age": 23,
    "email": "amit@yahoo.co.in"
},
{
    "id": 2,
    "name": "Vadik",
    "age": 24,
    "email": "vadik@yahoo.co.in"
},
{
    "id": 3,
    "name": "Prateek",
    "age": 21,
    "email": "prateek@yahoo.co.in"
},
{
    "id": 4,
    "name": "Harsh",
    "age": 25,
    "email": "harsh@yahoo.co.in"
},
{
    "id": 5,
    "name": "Swarit",
    "age": 24,
    "email": "Swarit@yahoo.co.in"
}
]
```

Enviamos una solicitud **GET** con la URL `http://localhost: 8080 / student / {id}`. Hemos invocado la URL `http://localhost: 8080 / student / 3`. Devuelve el detalle del alumno cuyo id es 3.



The screenshot shows a REST client interface. The top bar has a dropdown set to "GET" and a text input field containing the URL "http://localhost:8080/student/3". Below the URL are three tabs: "Pretty", "Raw", and "Preview". The "JSON" tab is selected, showing the following JSON response:

```
1 ▾ {|
2     "id": 3,
3     "name": "Prateek",
4     "age": 21,
5     "email": "prateek@yahoo.co.in"
6   }|
```

Del mismo modo, también podemos enviar una solicitud **DELETE**. Supongamos que queremos eliminar un registro de estudiante cuya identificación es 2.





Para eliminar un registro de estudiante, envíe una solicitud **DELETE** con la URL `http://localhost: 8080 / student / 2`. Vemos que el estudiante cuya identificación es **2** ha sido eliminado de la base de datos.

SELECT * FROM STUDENT;			
ID	AGE	EMAIL	NAME
1	23	amit@yahoo.co.in	Amit
3	21	prateek@yahoo.co.in	Prateek
4	25	harsh@yahoo.co.in	Harsh
5	24	Swarit@yahoo.co.in	Swarit

(4 rows, 22 ms)

[Descargar proyecto de ejemplo de base de datos H2](#)

## Operaciones Spring Boot CRUD

### ¿Qué es la operación CRUD?

La **ABM** es sinónimo de **creación, lectura / recuperar, actualizar y Borrar**. Estas son las cuatro funciones básicas del almacenamiento de persistencia.

La operación CRUD se puede definir como convenciones de interfaz de usuario que permiten ver, buscar y modificar información a través de formularios e informes basados en computadora. CRUD está orientado a datos y el uso estandarizado de **verbos de acción HTTP**. HTTP tiene algunos verbos importantes.

- **POST:** crea un nuevo recurso
- **GET:** Lee un recurso
- **PUT:** actualiza un recurso existente
- **DELETE:** elimina un recurso





Dentro de una base de datos, cada una de estas operaciones se asigna directamente a una serie de comandos. Sin embargo, su relación con una API RESTful es un poco más compleja.

## Operación CRUD estándar

- **Operación CREATE:** Realiza la instrucción INSERT para crear un nuevo registro.
- **Operación READ:** Lee los registros de la tabla en función del parámetro de entrada.
- **Operación UPDATE:** Ejecuta una declaración de actualización en la tabla. Se basa en el parámetro de entrada.
- **Operación DELETE:** Elimina una fila especificada en la tabla. También se basa en el parámetro de entrada.

## Cómo funcionan las operaciones CRUD

Las operaciones CRUD son la base de los sitios web más dinámicos. Por lo tanto, debemos diferenciar **CRUD** de los **verbos de acción HTTP**.

Supongamos que, si queremos **crear** un nuevo registro, deberíamos usar el verbo de acción **HTTP POST**. Para **actualizar** un registro, debemos usar el verbo **PUT**. Del mismo modo, si queremos **eliminar** un registro, debemos usar el verbo **DELETE**. A través de las operaciones CRUD, los usuarios y administradores tienen derecho a recuperar, crear, editar y eliminar registros en línea.

Tenemos muchas opciones para ejecutar operaciones CRUD. Una de las opciones más eficientes es crear un conjunto de procedimientos almacenados en SQL para ejecutar operaciones.

Las operaciones CRUD se refieren a todas las funciones principales que se implementan en aplicaciones de bases de datos relacionales. Cada letra del CRUD puede correlacionarse con una declaración SQL y métodos HTTP.

Operación	SQL	Verbos HTTP	Servicio web RESTful
-----------	-----	-------------	----------------------





<b>Crear</b>	INSERT	PUT / POST	POST
<b>Leer</b>	SELECT	GET	GET
<b>Actualizar</b>	UPDATE	PUT / POST / PATCH	PUT
<b>Borrar</b>	DELETE	DELETE	DELETE

## Spring Boot CrudRepository

Spring Boot proporciona una interfaz llamada **CrudRepository** que contiene métodos para operaciones CRUD. Está definido en el paquete **org.springframework.data.repository**. Extiende la interfaz del **repositorio** de datos de Spring . Proporciona una operación Crud genérica en un repositorio. Si queremos usar CrudRepository en una aplicación, tenemos que crear una interfaz y extender **CrudRepository** .

### Sintaxis

1. **public interface** CrudRepository<T, ID> **extends** Repository<T, ID>

dónde,

- o **T** es el tipo de dominio que administra el repositorio.
- o **ID** es el tipo de ID de la entidad que administra el repositorio.

Por ejemplo:

1. **public interface** StudentRepository **extends** CrudRepository<Student, Integer>
2. {
3. }





En el ejemplo anterior, hemos creado una interfaz llamada **StudentRepository** que extiende CrudRepository. Donde **Student** es el repositorio a administrar e **Integer** es el tipo de Id que se define en el repositorio de Student.

## Spring Boot JpaRepository

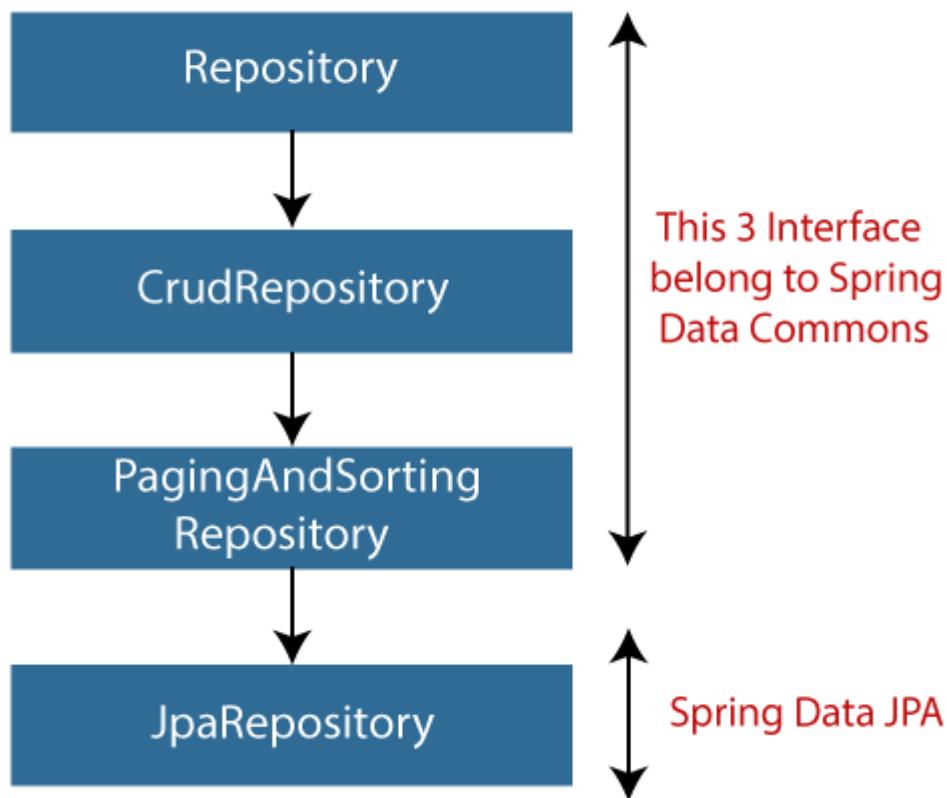
JpaRepository proporciona métodos relacionados con JPA, como vaciado, contexto de persistencia y elimina un registro en un lote. Está definido en el paquete **org.springframework.data.jpa.repository**. JpaRepository extiende tanto **CrudRepository** como **PagingAndSortingRepository**.

Por ejemplo:

1. **public interface** BookDAO **extends** JpaRepository
2. {
3. }



## Spring Data Repository Interface



### ¿Por qué deberíamos utilizar estas interfaces?

- Las interfaces permiten que Spring encuentre la interfaz del repositorio y cree objetos proxy para eso.
- Proporciona métodos que nos permiten realizar algunas operaciones habituales. También podemos definir métodos personalizados.

### CrudRepository frente a JpaRepository

CrudRepository	JpaRepository
CrudRepository no proporciona ningún método de paginación y clasificación.	JpaRepository extiende PagingAndSortingRepository. Proporciona todos los métodos para implementar la paginación.





Funciona como una interfaz de <b>marcador</b> .	JpaRepository extiende tanto <b>CrudRepository</b> como <b>PagingAndSortingRepository</b> .
Solo proporciona la función CRUD. Por ejemplo, <b>findById ()</b> , <b>findAll ()</b> , etc.	Proporciona algunos métodos adicionales junto con el método de PagingAndSortingRepository y CrudRepository. Por ejemplo, <b>flush ()</b> , <b>deleteInBatch ()</b> .
Se utiliza cuando no necesitamos las funciones proporcionadas por JpaRepository y PagingAndSortingRepository.	Se utiliza cuando queremos implementar la funcionalidad de paginación y clasificación en una aplicación.

## Ejemplo de operación Spring Boot CRUD

Configuremos una aplicación Spring Boot y realicemos la operación CRUD.

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

**Paso 2:** seleccione Spring Boot versión **2.3.0.M1**.

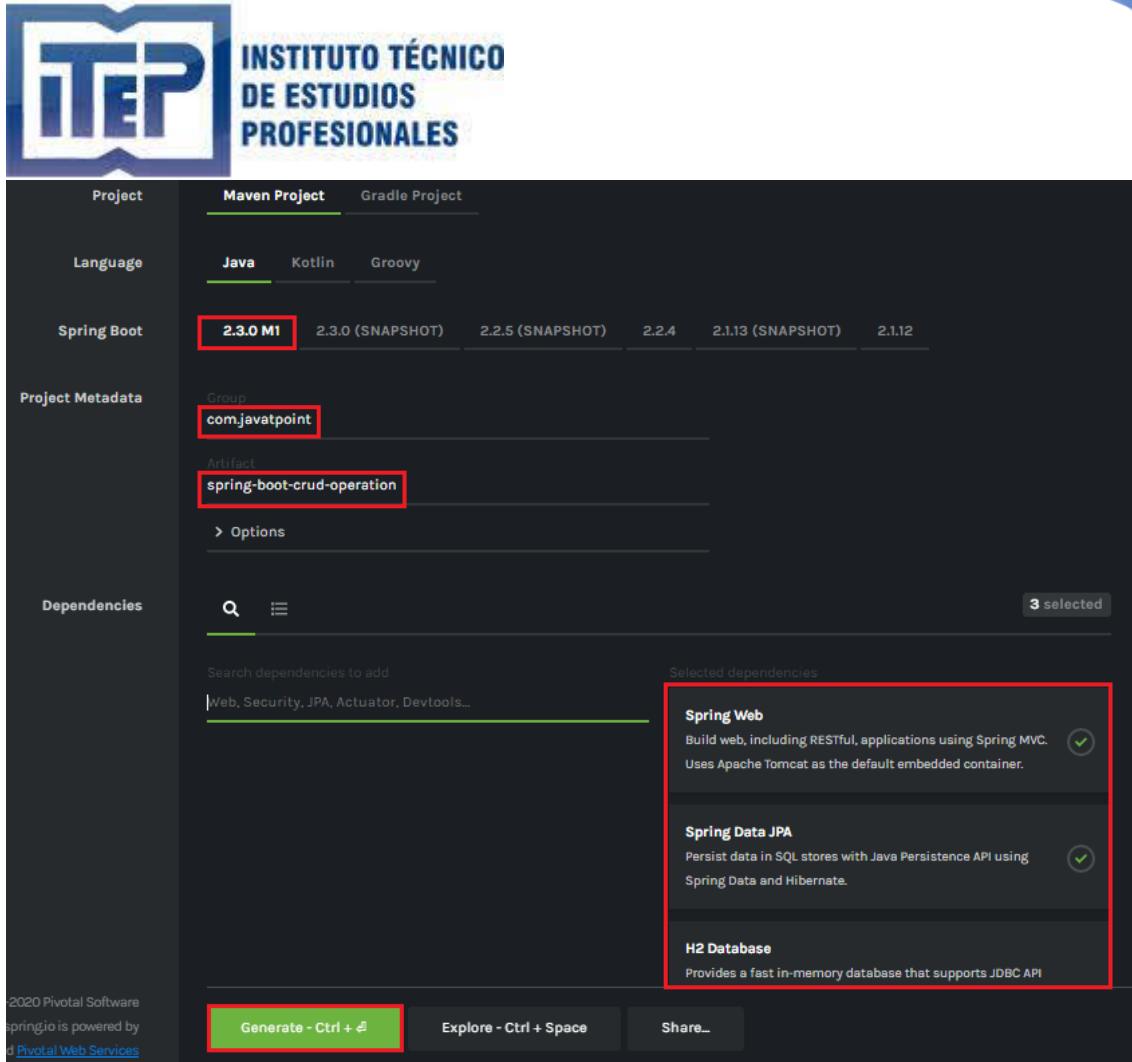
**Paso 2:** proporcione el nombre del **grupo**. Hemos proporcionado **com.javatpoint**.

**Paso 3:** proporcione el Id. Del **artefacto** . Hemos proporcionado **spring-boot-crud-operation**

**Paso 5:** agregue las dependencias **Spring Web**, **Spring Data JPA** y **H2 Database**.

**Paso 6:** Haga clic en el botón **Generar** . Cuando hacemos clic en el botón Generar, envuelve las especificaciones en un archivo **Jar** y lo descarga al sistema local.





**Paso 7:** extraiga el archivo Jar y péguelo en el espacio de trabajo de STS.

**Paso 8: Importe** la carpeta del proyecto a STS.

Archivo -> Importar -> Proyectos Maven existentes -> Examinar -> Seleccione la carpeta `spring-boot-crud-operation` -> Finalizar

La importación lleva algún tiempo.

**Paso 9:** Cree un paquete con el nombre **com.javatpoint.model** en la carpeta **src / main / java**.

**Paso 10:** Cree una clase de modelo en el paquete **com.javatpoint.model**. Hemos creado una clase modelo con el nombre **Books**. En la clase de Libros, hemos hecho lo siguiente:

- o Defina cuatro variables **bookid**, **bookname**, **author** y





- Genere Getters y Setters.
- Haga clic derecho en el archivo -> Fuente -> Generar Getters y Setters.
- Marque la clase como una **entidad** utilizando la anotación **@Entity**.
- Marque la clase como Nombre de **tabla** usando la anotación **@Table**.
- Defina cada variable como **Columna** utilizando la anotación **@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. }
```

**Paso 11:** Cree un paquete con el nombre **com.javatpoint.controller** en la carpeta **src / main / java**.





**Paso 12:** Cree una clase de controlador en el paquete **com.javatpoint.controller**. Hemos creado una clase de controlador con el nombre **BooksController**. En la clase BooksController, hemos hecho lo siguiente:

- Marque la clase como **RestController** usando la anotación **@RestController**.
- Conecte automáticamente la clase **BooksService** mediante la anotación **@Autowired**.
- Defina los siguientes métodos:
  - **getAllBooks ()**: devuelve una lista de todos los libros.
  - **getBooks ()**: Devuelve un detalle del libro que hemos especificado en la variable de ruta. Hemos pasado bookid como argumento usando la anotación **@PathVariable**. La anotación indica que un parámetro de método debe estar vinculado a una variable de plantilla de URI.
  - **deleteBook ()**: Elimina un libro específico que hayamos especificado en la variable de ruta.
  - **saveBook ()**: Guarda el detalle del libro. La anotación **@RequestBody** indica que un parámetro de método debe estar vinculado al cuerpo de la solicitud web.
  - **update ()**: el método actualiza un registro. Debemos especificar el registro en el cuerpo, que queremos actualizar. Para lograr lo mismo, hemos utilizado la anotación **@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 data
   base
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. }
```

**Paso 13:** Cree un paquete con el nombre **com.javatpoint.service** en la carpeta **src / main / java**.

**Paso 14:** Cree una clase de **servicio**. Hemos creado una clase de servicio con el nombre **BooksService** en el paquete **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 CrudReposit
    ory
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 CrudReposit
    ory
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 CrudRep
    ository
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. }
```



**Paso 15:** Cree un paquete con el nombre **com.javatpoint.repository** en la carpeta **src / main / java**.

**Paso 16:** Cree una interfaz de **repositorio**. Hemos creado una interfaz de repositorio con el nombre **BooksRepository** en el paquete **com.javatpoint.repository**. Extiende la interfaz de **Crud Repository**.

### **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, Integ-
er>
6. {
7. }
```

Ahora configuraremos la **URL de** la fuente de datos , **el nombre de la clase del controlador, el nombre de usuario y la contraseña** en el archivo **application.properties** .

**Paso 17:** Abra el archivo **application.properties** y configure las siguientes propiedades.

### **application.properties**

```

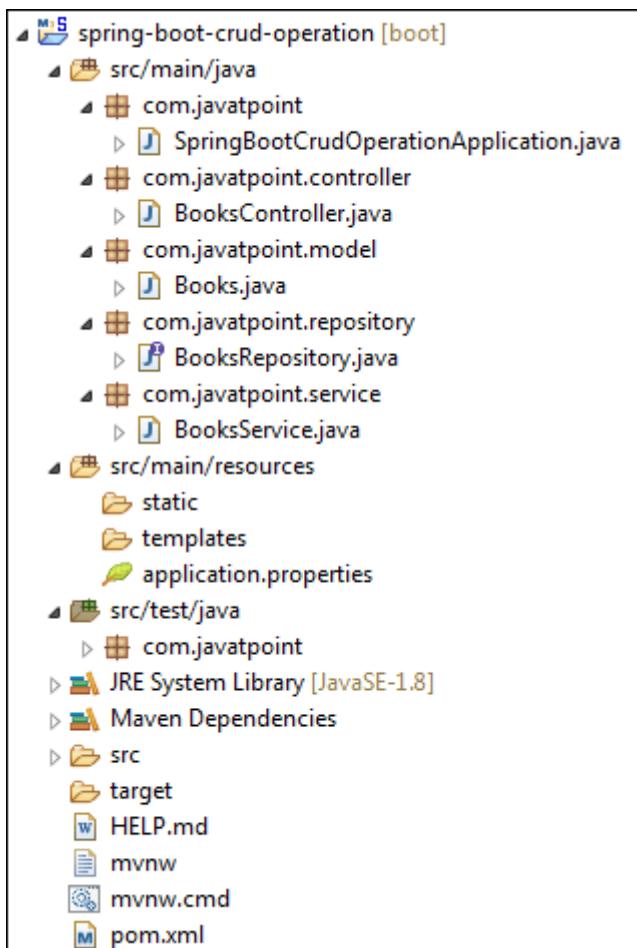
1. spring.datasource.url=jdbc:h2:mem:books_data
2. spring.datasource.driverClassName=org.h2.Driver
3. spring.datasource.username=sa
4. spring.datasource.password=
5. spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
6. #enabling the H2 console
7. spring.h2.console.enabled=true
```





**Nota:** No olvide habilitar la consola H2.

Después de crear todas las clases y paquetes, el directorio del proyecto tiene el siguiente aspecto.



Ahora ejecutaremos la aplicación.

**Paso 18:** Abra el archivo **SpringBootCrudOperationApplication.java** y ejecútelo como una aplicación Java.

### 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.

```

**Nota:** En los siguientes pasos usaremos rest client Postman. Por lo tanto, asegúrese de que la aplicación Postman ya esté instalada en su sistema.

**Paso 19:** Abra **postman** y haga lo siguiente:

- Seleccione el **POST**
- Invoque la URL `http://localhost: 8080 / books`.
- Seleccione el **body**
- Seleccione el **JSON de tipo de contenido (aplicación / json)**.
- Inserte los datos. Hemos insertado los siguientes datos en el Cuerpo:

```

1. {
2.   "bookid": "5433",
3.   "bookname": "Core and Advance Java",
4.   "author": "R. Nageswara Rao",
5.   "price": "800"
6. }

```

- Haga clic en **Enviar**

Cuando la solicitud se ejecuta con éxito, muestra el **Estado: 200 OK**. Significa que el registro se ha insertado correctamente en la base de datos.

Del mismo modo, hemos insertado los siguientes datos.



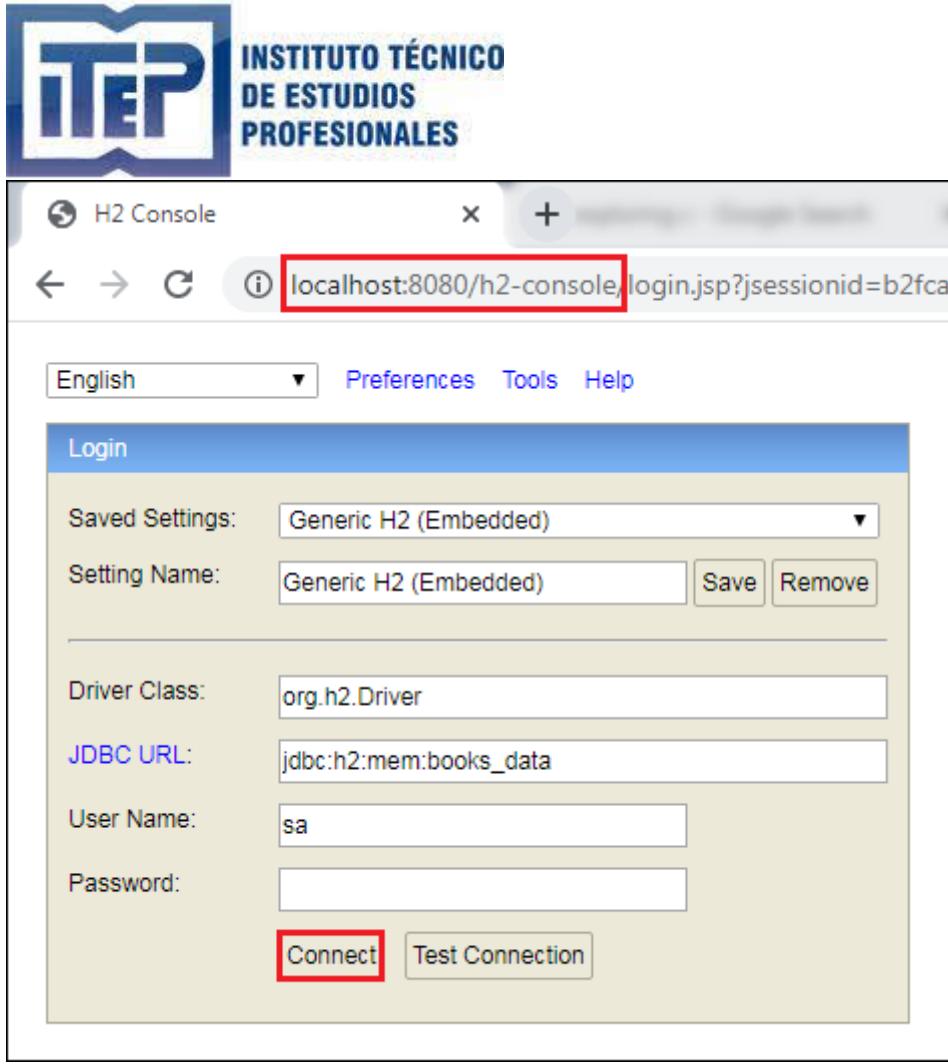


```
1. {  
2.   "bookid": "0982",  
3.   "bookname": "Programming with Java",  
4.   "author": "E. Balagurusamy",  
5.   "price": "350"  
6. }  
7. {  
8.   "bookid": "6321",  
9.   "bookname": "Data Structures and Algorithms in Java",  
10.  "author": "Robert Lafore",  
11.  "price": "590"  
12.}  
13. {  
14.   "bookid": "5433",  
15.   "bookname": "Effective Java",  
16.   "author": "Joshua Bloch",  
17.   "price": "670"  
18.}
```

Accedamos a la consola H2 para ver los datos.

**Paso 20:** Abra el navegador e invoque la URL `http://localhost:8080/h2-console`. Haga clic en el botón **Conectar**, como se muestra a continuación.





Después de hacer clic en el botón **Conectar**, vemos la tabla **books** en la base de datos, como se muestra a continuación.

The screenshot shows the H2 Database browser interface. On the left, there's a tree view of database objects. A red box highlights the 'BOOKS' table under the 'BOOKS' schema. Other visible objects include 'INFORMATION\_SCHEMA' and 'Users'. At the bottom, there's a note about the version: 'H2 1.4.200 (2019-10-14)'.

**Paso 21:** Haga clic en la tabla **books** y luego haga clic en el botón **Ejecutar**. La tabla muestra los datos que hemos insertado en el cuerpo.





Run Run Selected Auto complete Clear SQL statement:

```
SELECT * FROM BOOKS
```

SELECT \* FROM BOOKS BOOKS;

BOOKID	AUTHOR	BOOKNAME	PRICE
982	E. Balagurusamy	Programming with Java	350
5433	R. Nageswara Rao	Core and Advance Java	800
6321	Robert Lafore	Data Structures and Algorithms in Java	590
6830	Yashavant Kanetkar	Exploring C	670

(4 rows, 23 ms)

[Edit](#)

**Paso 22:** Abra **Postman** y envíe una solicitud **GET** con la URL `http://localhost:8080 / books`. Devuelve los datos que hemos insertado en la base de datos.

```

[
  {
    "bookid": 982,
    "bookname": "Programming with Java",
    "author": "E. Balagurusamy",
    "price": 350
  },
  {
    "bookid": 5433,
    "bookname": "Core and Advance Java",
    "author": "R. Nageswara Rao",
    "price": 800
  },
  {
    "bookid": 6321,
    "bookname": "Data Structures and Algorithms in Java",
    "author": "Robert Lafore",
    "price": 590
  },
  {
    "bookid": 6830,
    "bookname": "Exploring C",
    "author": "Yashavant Kanetkar",
    "price": 670
  }
]

```





Enviamos una solicitud **GET** con la URL `http://localhost:8080/book/{bookid}`. Hemos especificado el **bookid 6830**. Devuelve el detalle del libro cuyo id es 6830.

The screenshot shows a POSTMAN interface. The method is set to **GET** and the URL is `http://localhost:8080/book/6830`. The **Authorization** tab is selected. In the **Body** tab, the **Type** is set to **No Auth**. The **JSON** response is displayed as:

```

1 → [
2   "bookid": 6830,
3   "bookname": "Exploring C",
4   "author": "Yashavant Kanetkar",
5   "price": 670
6 ]

```

Del mismo modo, también podemos enviar una solicitud **DELETE** para eliminar un registro. Supongamos que queremos eliminar un registro de libro cuyo id es **5433**.

Seleccione el método **DELETE** e invoque la URL `http://localhost:8080/book/5433`. Nuevamente, ejecute la consulta **Seleccionar** en la consola H2. Vemos que el libro cuyo id es **5433** ha sido eliminado de la base de datos.

SELECT * FROM BOOKS;			
BOOKID	AUTHOR	BOOKNAME	PRICE
982	E. Balagurusamy	Programming with Java	350
6321	Robert Lafore	Data Structures and Algorithms in Java	590
6830	Yashavant Kanetkar	Exploring C	670

(3 rows, 23 ms)

Del mismo modo, también podemos actualizar un registro enviando una solicitud **PUT**. Actualicemos el precio del libro cuyo id es **6321**.





- Seleccione el **PUT**
- En el cuerpo de la solicitud, pegue el registro que desea actualizar y realice los cambios. En nuestro caso, queremos actualizar el registro del libro cuyo id es 6321. En el siguiente registro, hemos cambiado el precio del libro.

```

1. {
2.   "bookid": "6321",
3.   "bookname": "Data Structures and Algorithms in Java",
4.   "author": "Robert Lafore",
5.   "price": "500"
6. }
```

- Haga clic en **Enviar**

Ahora, muévase a la consola H2 y vea que los cambios se han reflejado o no. Vemos que el precio del libro ha cambiado, como se muestra a continuación.

SELECT * FROM BOOKS;			
BOOKID	AUTHOR	BOOKNAME	PRICE
982	E. Balagurusamy	Programming with Java	350
6321	Robert Lafore	Data Structures and Algorithms in Java	500
6830	Yashavant Kanetkar	Exploring C	390

(3 rows, 20 ms)

[Descargar proyecto de operación CRUD](#)

## Spring Boot Thymeleaf

### ¿Qué es Thymeleaf?

El **thymeleaf** es una biblioteca de Java de código abierto que está disponible bajo la **licencia Apache 2.0**. Es un motor de plantillas **HTML5 / XHTML /**





**XML**. Es un motor de **plantillas Java del lado del servidor** para entornos web (basados en servlets) y no web (fuera de línea). Es perfecto para el desarrollo web HTML5 JVM de hoy en día. Proporciona una integración completa con Spring Framework.

Aplica un conjunto de transformaciones a los archivos de plantilla para mostrar datos o texto producido por la aplicación. Es apropiado para servir XHTML / HTML5 en aplicaciones web.

El objetivo de Thymeleaf es proporcionar una forma **elegante y bien formada** de crear plantillas. Se basa en etiquetas y atributos XML. Estas etiquetas XML definen la ejecución de la lógica predefinida en el DOM (Modelo de objeto de documento) en lugar de escribir explícitamente esa lógica como código dentro de la plantilla. Es un sustituto de **JSP**.

La arquitectura de Thymeleaf permite el **procesamiento rápido** de plantillas que depende del almacenamiento en caché de los archivos analizados. Utiliza la menor cantidad posible de operaciones de E / S durante la ejecución.

## ¿Por qué usamos Thymeleaf?

JSP es más o menos similar a HTML. Pero no es completamente compatible con HTML como Thymeleaf. Podemos abrir y mostrar un archivo de plantilla Thymeleaf normalmente en el navegador, mientras que el archivo JSP no lo hace.

Thymeleaf admite expresiones variables (`$ {...}`) como Spring EL y se ejecuta en los atributos del modelo, las expresiones de asterisco (`* {...}`) se ejecutan en el bean de respaldo del formulario, las expresiones hash (`# {...}`) son para internacionalización y expresiones de enlace (`@ {...}`) reescriben las URL.

Al igual que JSP, Thymeleaf funciona bien para correos electrónicos con HTML enriquecido.

## ¿Qué tipo de plantillas puede procesar Thymeleaf?

Thymeleaf puede procesar seis tipos de plantillas (también conocidas como **modo de plantilla**) que son las siguientes:

- XML
- XML válido





- XHTML
- XHTML válido
- HTML5
- HTML5 heredado

A excepción del modo Legacy HTML5, todos los modos anteriores se refieren a archivos **XML bien definidos**. Nos permite procesar archivos HTML5 con características como **etiquetas independientes, atributos de etiquetas sin valor o no escritos entre comillas**.

Para procesar archivos en este modo específico, Thymeleaf realiza una transformación que convierte los archivos en un archivo **XML bien formado** (archivo HTML5 válido).

**Nota:** En Thymeleaf, la validación solo está disponible para plantillas XHTML y XML.

Thymeleaf también nos permite definir nuestro propio modo especificando ambas formas de analizar plantillas en este modo. De esta manera, todo lo que se pueda modelar como un árbol DOM podría ser procesado efectivamente como una plantilla por Thymeleaf.

## El dialecto estándar

Thymeleaf es un framework de motor de plantillas que nos permite definir los nodos DOM. Los nodos DOM procesados en las plantillas.

Un objeto que aplica lógica a un nodo DOM se llama **procesador**. Un conjunto de procesadores, junto con algunos artefactos adicionales, se llama **dialecto**. El dialecto que contiene la biblioteca principal de Thymeleaf se llama **Dialecto estándar**.

Si queremos definir nuestra propia lógica de procesamiento mientras aprovechamos las funciones avanzadas de la biblioteca, podemos definir nuestros propios dialectos. En un motor de plantillas, podemos configurar varios dialectos a la vez.

Los paquetes de integración de Thymeleaf (thymeleaf-spring3 y thymeleaf-spring4) definen un dialecto llamado **SpringStandard Dialect**. El dialecto estándar y SpringStandard son casi iguales. Pero el dialecto estándar tiene





algunos pequeños cambios que hacen un mejor uso de algunas características en Spring Framework.

Por ejemplo, use Spring Expression Language en lugar del estándar ONGL (Object-Graph Navigation Language) de Thymeleaf.

El dialecto estándar puede procesar la plantilla en cualquier modo. Pero es perfecto para los modos de plantilla orientados a la web (HTML5 y XHTML). Admite y valida las siguientes especificaciones XHTML:

- XHTML 1.0 Transicional
- XHTML 1.0 estricto
- Conjunto de marcos XHTML 1.0
- XHTML 1.1.

El procesador de dialecto estándar es el procesador de atributos que permite a los navegadores mostrar archivos de plantilla HTML5 / XHTML antes de ser procesados. Es porque ignoran los atributos adicionales.

Por ejemplo, cuando un archivo JSP usa una biblioteca de etiquetas, incluye un fragmento de código que un navegador no puede visualizar como:

1. **<form:inputText name="student.Name" value="\${student.name}" />**

El dialecto estándar de Thymeleaf nos permite lograr la misma funcionalidad con el siguiente código.

1. **<input type="text" name="student Name" value="Thomas" th:value="\${student.name}" />**

El código anterior también nos permite definir un atributo de **value** en él (**Thomas**). El valor se mostrará cuando se abra el prototipo en el navegador. El atributo será sustituido por el valor resultante de la evaluación de  **\${student.name}** durante el procesamiento de Thymeleaf de la plantilla.





Permite que un diseñador y un desarrollador trabajen en el mismo archivo de plantilla, lo que reduce los esfuerzos necesarios para transformar un prototipo estático en un archivo de plantilla funcional. Se conoce como **plantilla natural**.

## Características de Thymeleaf

- Funciona tanto en entornos web como no web.
- Motor de plantillas Java para HTML5 / XML / XHTML.
- Su caché de plantillas analizadas de alto rendimiento reduce las E / S al mínimo.
- Se puede utilizar como marco de motor de plantilla si es necesario.
- Admite varios modos de plantilla: XML, XHTML y HTML5.
- Permite a los desarrolladores ampliar y crear dialectos personalizados.
- Se basa en conjuntos de características modulares llamados dialectos.
- Apoya la internacionalización.

## Implementación de Thymeleaf

Podemos implementar el motor de plantillas Thymeleaf agregando la dependencia **spring-boot-starter-thymeleaf** en el archivo pom.xml de nuestra aplicación. Spring Boot configura el motor de plantilla para leer el archivo de plantilla de **/ resource / templates**.

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

## Ejemplo de Spring Boot Thymeleaf

Creemos una aplicación Spring Boot e implementemos la plantilla Thymeleaf.

**Paso 1:** Abra Spring Initializr <http://start.spring.io> .

**Paso 2:** seleccione Spring Boot versión **2.3.0.M1**.

**Paso 2:** proporcione el nombre del **grupo** . Hemos proporcionado **com.javatpoint** .





**Paso 3:** proporcione el Id. Del **artefacto**. Hemos proporcionado **spring-boot-thymeleaf-view-example**.

**Paso 5:** agregue las dependencias **Spring Web** y **Thymeleaf**.

**Paso 6:** Haga clic en el botón **Generar**. Cuando hacemos clic en el botón Generar, envuelve las especificaciones en un archivo **Jar** y lo descarga al sistema local.

The screenshot shows the Spring Initializr interface. The 'Project' dropdown is set to 'Maven Project'. The 'Language' dropdown is set to 'Java'. Under 'Spring Boot', '2.3.0.M1' is selected. In the 'Project Metadata' section, the 'Group' field contains 'com.javatpoint' and the 'Artifact' field contains 'spring-boot-thymeleaf-view-example'. In the 'Dependencies' section, 'Spring Web' and 'Thymeleaf' are selected and highlighted with a red box. The 'Generate - Ctrl + A' button at the bottom is also highlighted with a red box.

**Paso 7: extraiga** el archivo Jar y péguelo en el espacio de trabajo de STS.

**Paso 8: Importe** la carpeta del proyecto en STS.

Archivo -> Importar -> Proyectos Maven existentes -> Examinar -> Seleccione la carpeta `spring-boot-thymeleaf-view-example` -> Finalizar

La importación lleva algún tiempo.





**Paso 9:** cree una clase de modelo en el paquete **com.javatpoint** . Hemos creado una clase modelo con el nombre **User**.

En esta clase, hemos definido dos variables **name** y **email** y generamos **Getters** y **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.}
```

**Paso 10:** crea una clase de controlador. Hemos creado una clase de controlador con el nombre **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. }
```

En el siguiente paso, crearemos las plantillas Thymeleaf.

**Paso 11:** Dentro de la carpeta **templates** (src / main / resources / templates) del proyecto, cree una plantilla Thymeleaf con el nombre **user-data** .

Haga clic con el botón derecho en la carpeta de plantillas -> Nuevo -> Otro -> Archivo HTML -> Siguiente -> Proporcione el nombre del archivo -> Finalizar

**Nota: No olvide implementar lo siguiente en el archivo de plantilla.**

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>
```

**Paso 12:** De manera similar, cree un archivo **HTML** en las plantillas de carpeta. Hemos creado un archivo HTML con el nombre **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>
```





INSTITUTO TÉCNICO  
DE ESTUDIOS  
PROFESIONALES

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>

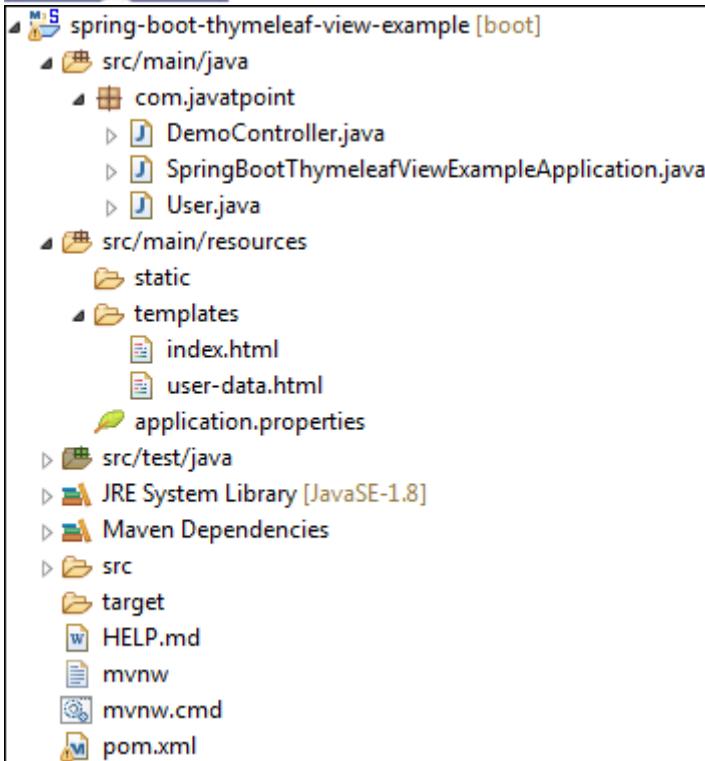
**Paso 13:** Abra el archivo **application.properties** y agregue las siguientes propiedades en él.

#### **application.properties**

1. spring.thymeleaf.cache=false
2. spring.thymeleaf.suffix: .html

Después de crear todos los archivos, carpetas y paquetes, el directorio del proyecto tiene el siguiente aspecto:





Ejecutemos la aplicación.

**Paso 14:** Abra el archivo **SpringBootThymeleafViewExampleApplication.java** y ejecútelo como aplicación Java.

### SpringBootThymeleafViewExampleApplication.java

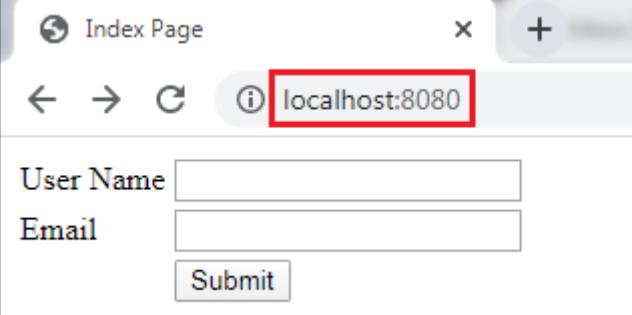
```

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
10.   }
11. }

```

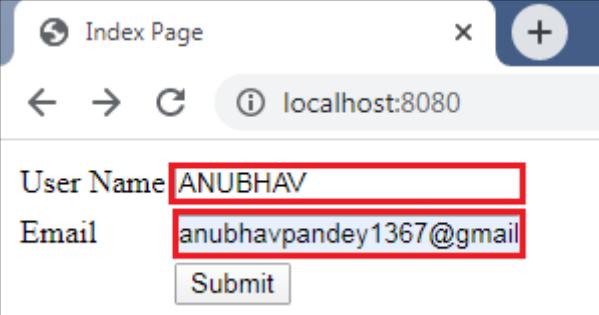


**Paso 15:** Ahora, abra el navegador e invoque la URL `http:// localhost: 8080`. Muestra la salida, como se muestra a continuación.



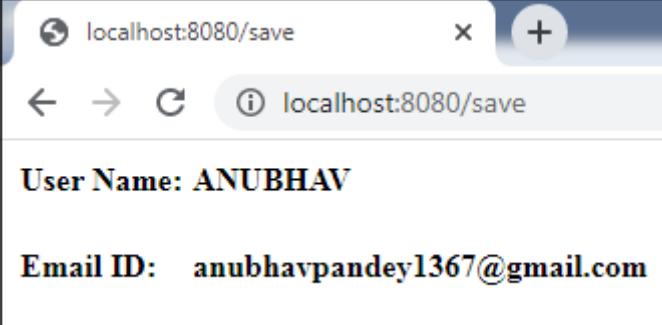
The screenshot shows a browser window titled "Index Page". The address bar contains the URL "localhost:8080", which is highlighted with a red box. Below the address bar, there are two input fields: "User Name" and "Email", both empty. At the bottom of the form is a "Submit" button.

Proporcione el **nombre de usuario** y el **correo electrónico** y haga clic en el botón **Enviar**.



The screenshot shows the same browser window as before, but now the "User Name" field contains the text "ANUBHAV" and the "Email" field contains the text "anubhavpandey1367@gmail.com", both of which are highlighted with red boxes.

Después de hacer clic en el botón **Enviar**, la URL cambia a `http:// localhost: 8080 / save` y muestra los datos del usuario, como se muestra a continuación.



The screenshot shows a browser window with the URL "localhost:8080/save". The page content displays the submitted user data: "User Name: ANUBHAV" and "Email ID: anubhavpandey1367@gmail.com".

En esta sección, hemos discutido la vista Thymeleaf. Si queremos que la vista sea más atractiva, podemos agregar archivos **CSS** y **JS** en la aplicación. Estos archivos deben estar ubicados en la carpeta **src / main / resources / static**.





INSTITUTO TÉCNICO  
DE ESTUDIOS  
PROFESIONALES

[Descargar proyecto de ejemplo de Thymeleaf View](#)

