Spring Boot

Autor: Rafael Alcocer Caldera

Introducción: Spring Boot

Fecha: 16-May-2021

Tabla de Contenido

Spring Boot

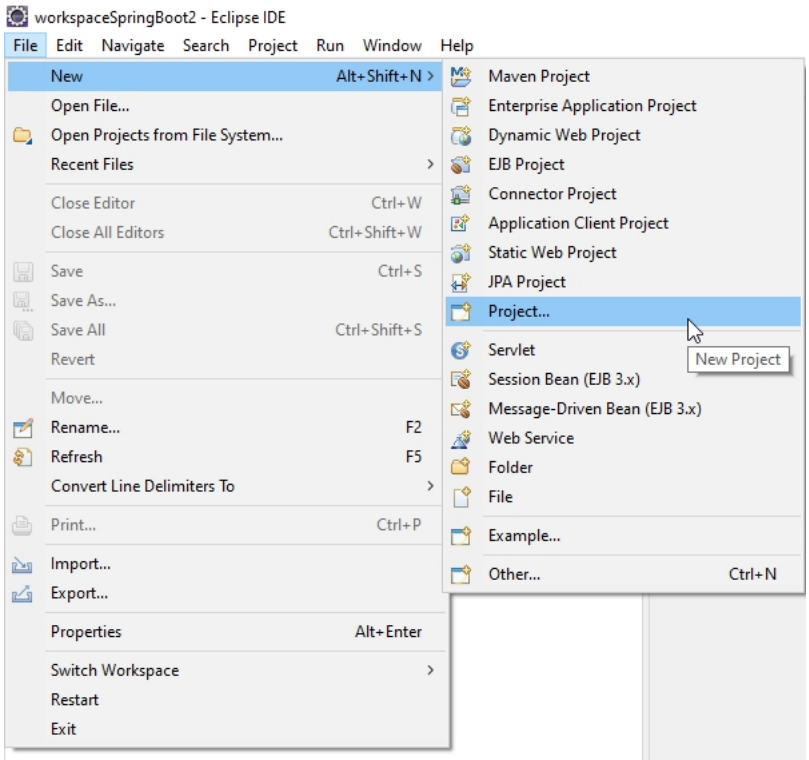
- Spring Boot lets you create Spring Enterprise Applications from scratch.

What is Spring Boot?
Spring Boot makes it easy to create standalone, production-grade, Spring based Applications that you can "just run".

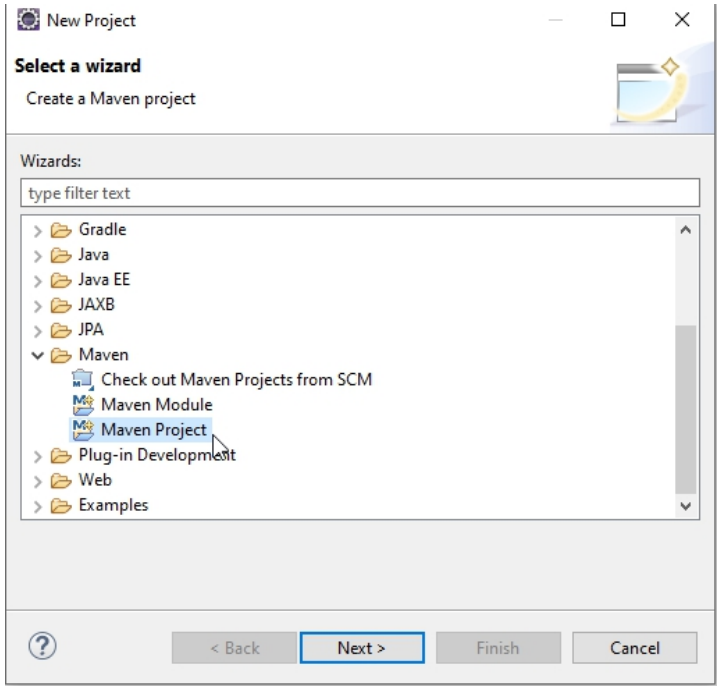
Features:
- Opionated
- Convention over configuration
- Standalone
- Production ready

Crear Proyecto Maven

File -> New -> Project...

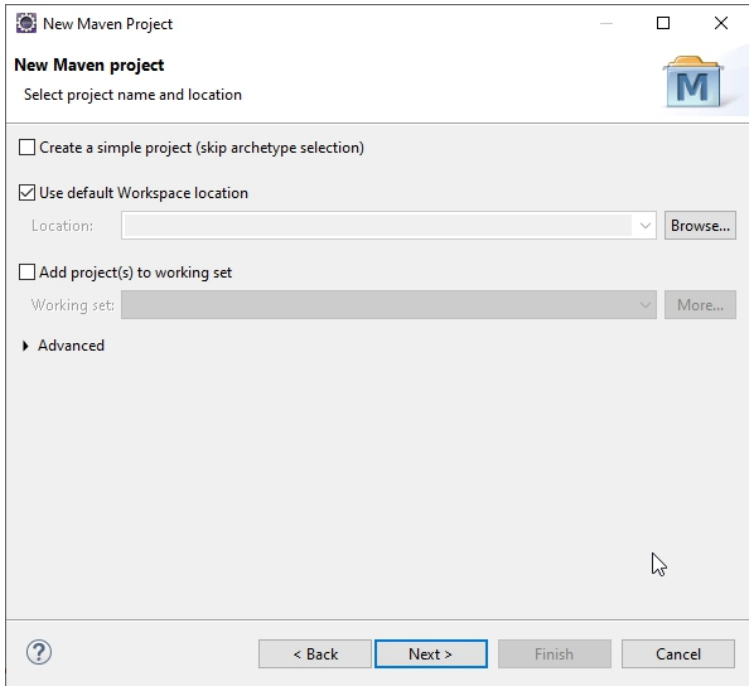


En la parte superior vemos **Maven Project**, podríamos seleccionar este y listo, pero si no aparece, entonces al seleccionar **Project...**

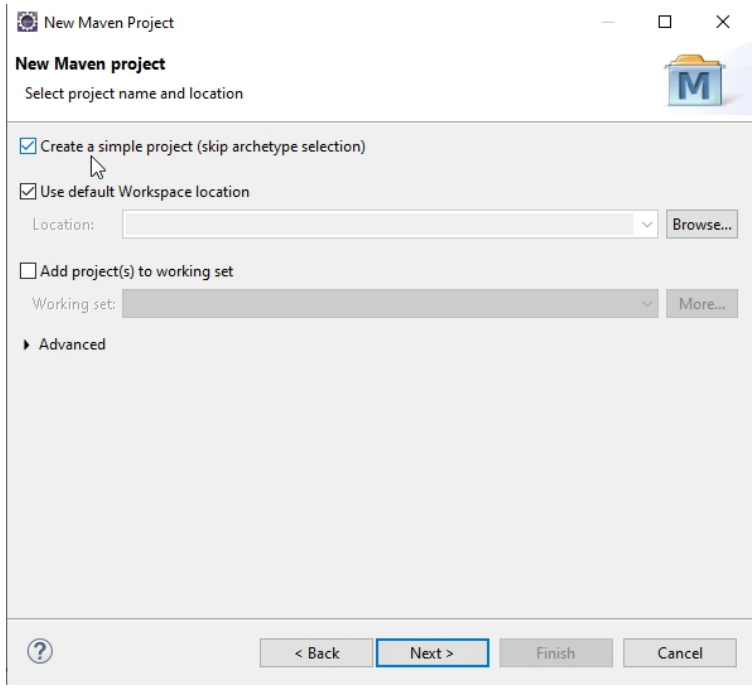


Nos aparece este wizard, seleccionamos **Maven -> Maven Project**

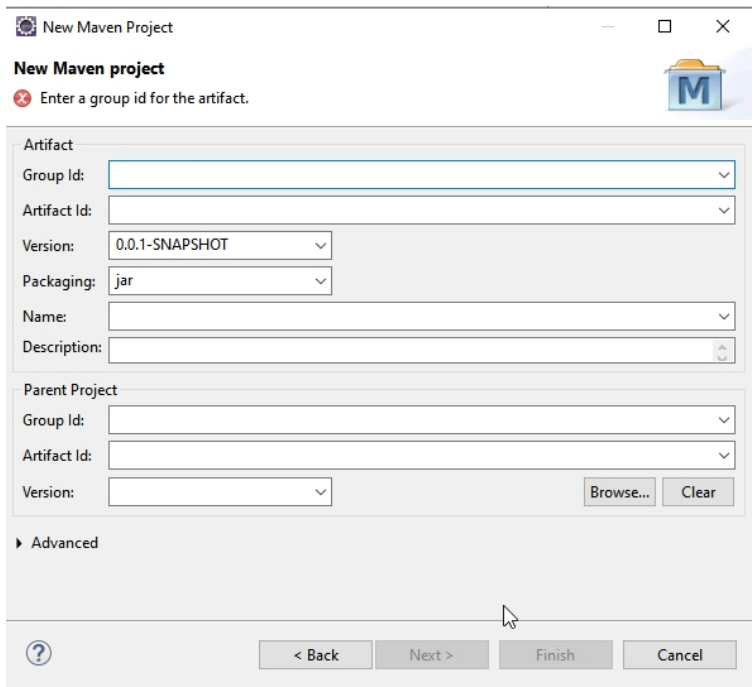
Se da clic en **Next>**



Seleccionamos la casilla de verificación que dice **Create a simple project**

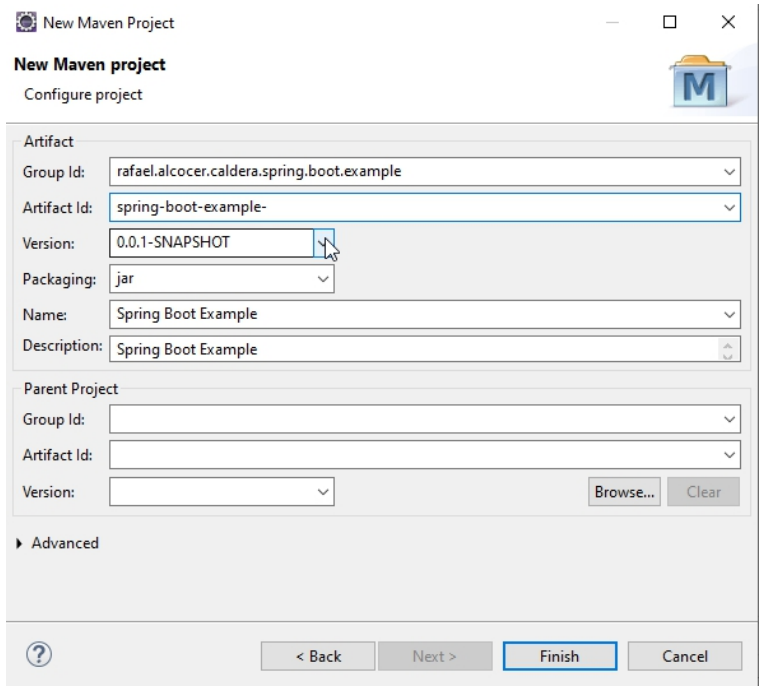


Se da clic en **Next>**

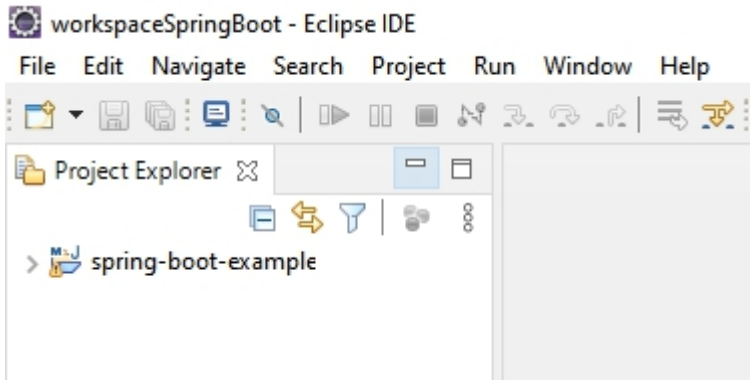


Llenamos con los siguientes datos:

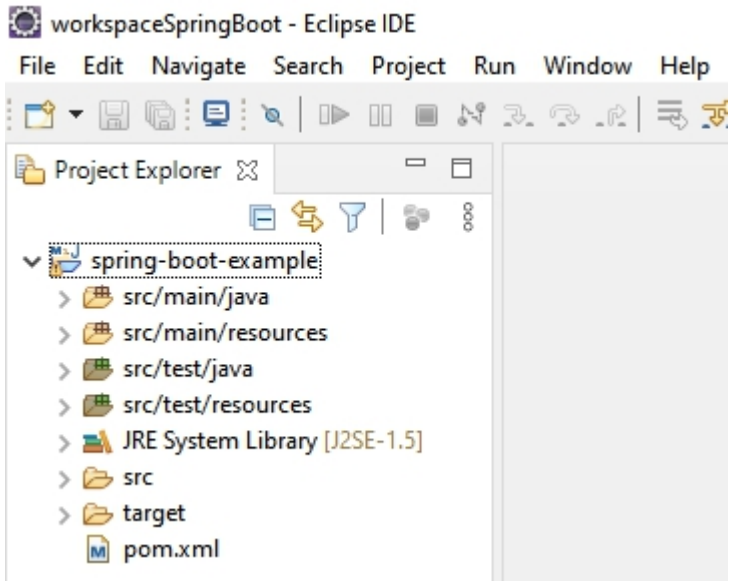
- Group Id: rafael.alcocer.caldera.spring.boot.example
- Artifact Id: spring-boot-example
- Name: Spring Boot Example
- Description: Spring Boot Example



En esta misma sección se puede poner la información del Parent Project, pero lo haremos editando el archivo **pom.xml**. Así que aquí damos clic en **Finish**.



Expandimos el proyecto



Damos doble clic en **pom.xml**:

pom.xml

A lo anterior le agregamos <parent>, <properties> y <dependencies>:

```
<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>
    <groupId>rafael.alcocer.caldera.spring.boot.example</groupId>
    <artifactId>spring-boot-example</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>Spring Boot Example</name>
    <description>Spring Boot Example</description>

    <properties>
        <java-version>1.8</java-version>
    </properties>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.0</version>
    </parent>

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

</project>
```

<dependencies> nos dice qué JARs bajar y

<parent> nos dice que versiones de esos JARs bajar

Bill of materials

La lista de posibles combinaciones de JARs que trabajen sin problemas.

No nos necesitamos preocupar de versiones individuales, únicamente del <parent> y esto por sí mismo instruye a Spring o Maven sobre las posibles combinaciones de versiones que trabajen bien.

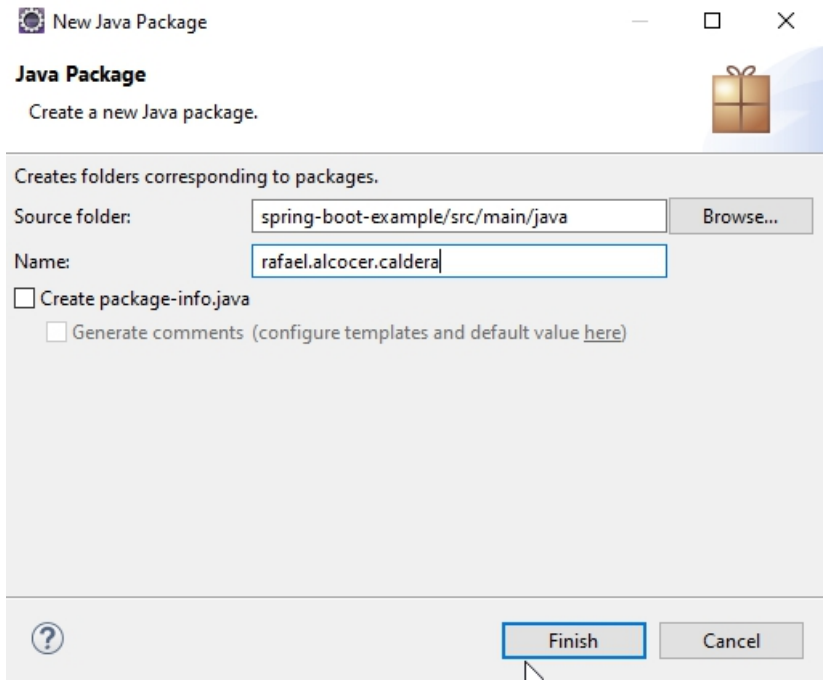
Starting Spring Boot

- Sets up default configuration
- Starts Spring application context
- Performs class path scan
- Starts Tomcat server

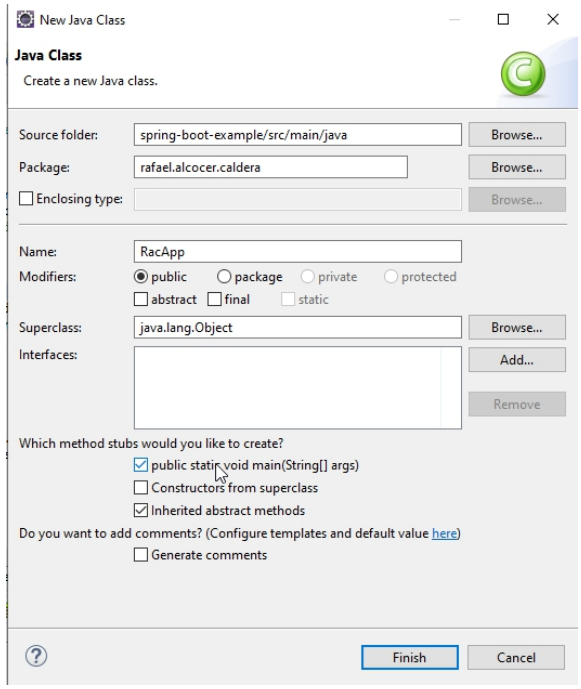
Embbeded Tomcat Server

- Convenience
- Servlet container config is now application config
- Standalone application
- Useful for microservices architecture

Creamos un nuevo paquete:



Creamos una clase que nos va a servir de **BootStrap**:



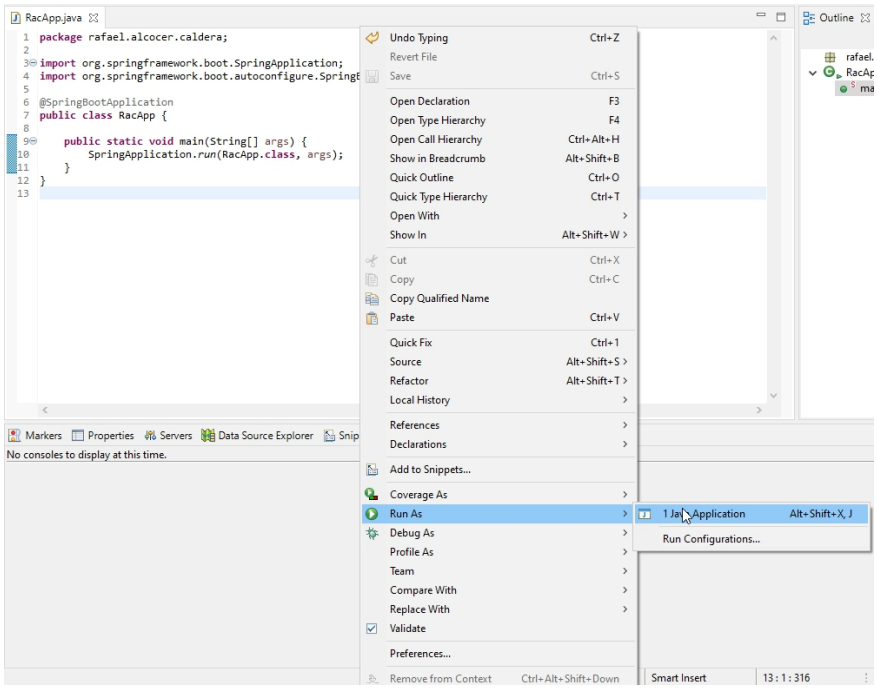
BootStrap

A technique of loading a program into a computer by means of a few initial instructions that enable the introduction of the rest of the program from an input device.

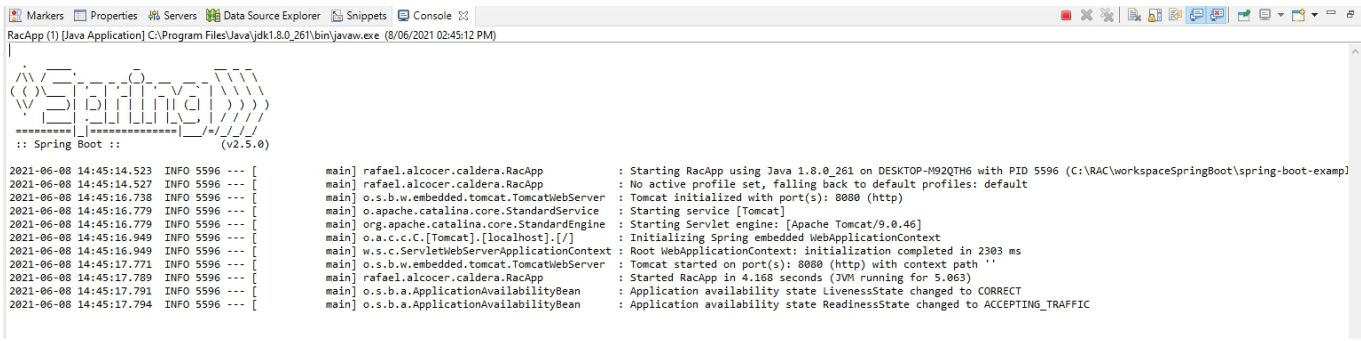
La clase finalmente queda así:

```
package rafael.alcocer.caldera;

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

@SpringBootApplication
public class RacApp {

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

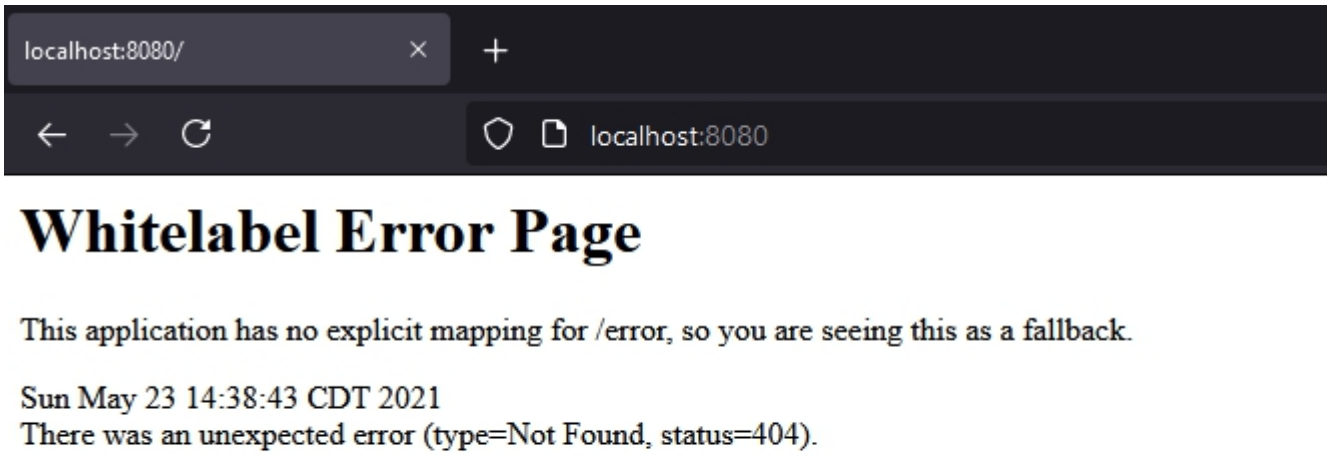
Si corremos la aplicación:



Podemos verificar en la consola que el Contenedor Web Tomcat inicia correctamente:



Si se ingresa a la URL obtendremos esto:



@SpringBootApplication

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Inherited
 @SpringBootConfiguration
 @EnableAutoConfiguration
 @ComponentScan(excludeFilters={@ComponentScan.Filter(type=CUSTOM,classes=TypeExcludeFilter.class),})
public @interface SpringBootApplication
```

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes.

Por lo tanto, esto:

```
@SpringBootApplication
public class RacApp {...}
```

Es lo mismo que esto:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class RacApp {...}
```

@Configuration

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Component
public @interface Configuration
```

Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime, for example:

```
@Configuration
public class AppConfig {

    @Bean
    public MyBean myBean() {
        // instantiate, configure and return bean ...
    }
}
```

@Configuration is meta-annotated with **@Component**, therefore @Configuration classes are candidates for component scanning (typically using Spring XML's <context:component-scan/> element) and therefore may also take advantage of @Autowired/@Inject like any regular @Component. In particular, if a single constructor is present autowiring semantics will be applied transparently for that constructor:

@EnableAutoConfiguration

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Inherited
 @AutoConfigurationPackage
 @Import(value=AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration
```

Enable auto-configuration of the Spring Application Context, attempting to guess and configure beans that you are likely to need. Auto-configuration classes are usually applied based on your classpath and what beans you have defined.
For example, if you have tomcat-embedded.jar on your classpath you are likely to want a TomcatServletWebServerFactory (unless you have defined your own ServletWebServerFactory bean).

When using @SpringBootApplication, the auto-configuration of the context is automatically enabled and adding this annotation has therefore no additional effect.

Auto-configuration tries to be as intelligent as possible and will back-away as you define more of your own configuration. You can always manually exclude() any configuration that you never want to apply (use excludeName() if you don't have access to them). You can also exclude them via the spring.autoconfigure.exclude property.
Auto-configuration is always applied after user-defined beans have been registered.

The package of the class that is annotated with @EnableAutoConfiguration, usually via @SpringBootApplication, has specific significance and is often used as a 'default'. For example, it will be used when scanning for @Entity classes. It is generally recommended that you place @EnableAutoConfiguration (if you're not using @SpringBootApplication) in a root package so that all sub-packages and classes can be searched.

Auto-configuration classes are regular Spring @Configuration beans. They are located using the SpringFactoriesLoader mechanism (keyed against this class). Generally auto-configuration beans are @Conditional beans (most often using @ConditionalOnClass and @ConditionalOnMissingBean annotations).

@ComponentScan

```
@Retention(value=RUNTIME)
 @Target(value=TYPE)
 @Documented
 @Repeatable(value=ComponentScans.class)
public @interface ComponentScan
```

Configures component scanning directives for use with @Configuration classes. Provides support parallel with Spring XML's <context:component-scan> element.

Either basePackageClasses() or basePackages() (or its alias value()) may be specified to define specific packages to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

Note that the <context:component-scan> element has an annotation-config attribute; however, this annotation does not. This is because in almost all cases when using @ComponentScan, default annotation config processing (e.g. processing @Autowired and friends) is assumed.
Furthermore, when using AnnotationConfigApplicationContext, annotation config processors are always registered, meaning that any attempt to disable them at the @ComponentScan level would be ignored.

https://www.baeldung.com/spring-component-scanning
With Spring, we use the **@ComponentScan** annotation along with the **@Configuration** annotation to specify the packages that we want to be scanned.
@ComponentScan without arguments tells Spring to scan the current package and all of its sub-packages.

Let's say we have the following @Configuration in **com.baeldung.componentscan.springapp** package:

```
@Configuration
@ComponentScan
public class SpringComponentScanApp {
    private static ApplicationContext applicationContext;

    @Bean
    public ExampleBean exampleBean() {
        return new ExampleBean();
    }

    public static void main(String[] args) {
        applicationContext =
          new AnnotationConfigApplicationContext(SpringComponentScanApp.class);

        for (String beanName : applicationContext.getBeanDefinitionNames()) {
            System.out.println(beanName);
        }
    }
}
```

In addition, we have the Cat and Dog components in **com.baeldung.componentscan.springapp.animals** package:

```
package com.baeldung.componentscan.springapp.animals;
// ...
@Component
public class Cat {}
```

```
package com.baeldung.componentscan.springapp.animals;
// ...
@Component
public class Dog {}
```

Finally, we have the Rose component in **com.baeldung.componentscan.springapp.flowers** package:

```
package com.baeldung.componentscan.springapp.flowers;
// ...
@Component
public class Rose {}
```

The output of the main() method will contain all the beans of **com.baeldung.componentscan.springapp** package and its sub-packages:

- springComponentScanApp
- cat
- dog
- rose
- exampleBean

Note that the main application class is also a bean, as it's annotated with @Configuration, which is a @Component.

Hello REST Endpoint

Vamos a crear en endpoint que nos regrese un Hello... al momento de ingresar a la URL.

Sobre el mismo proyecto anterior vamos a crear una clase

```
package com.rac.controller;

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

@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String hello() {
        return "Hello...";
    }
}
```

@RestController

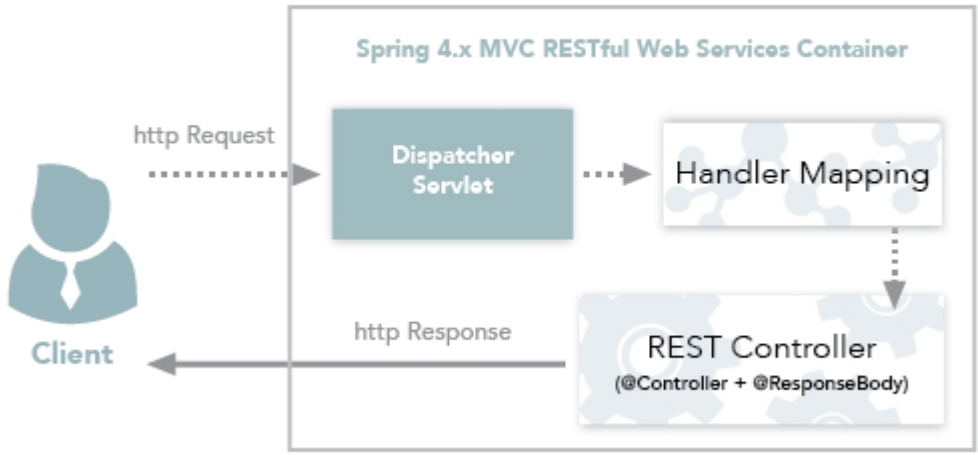
```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Controller
 @ResponseBody
public @interface RestController
```

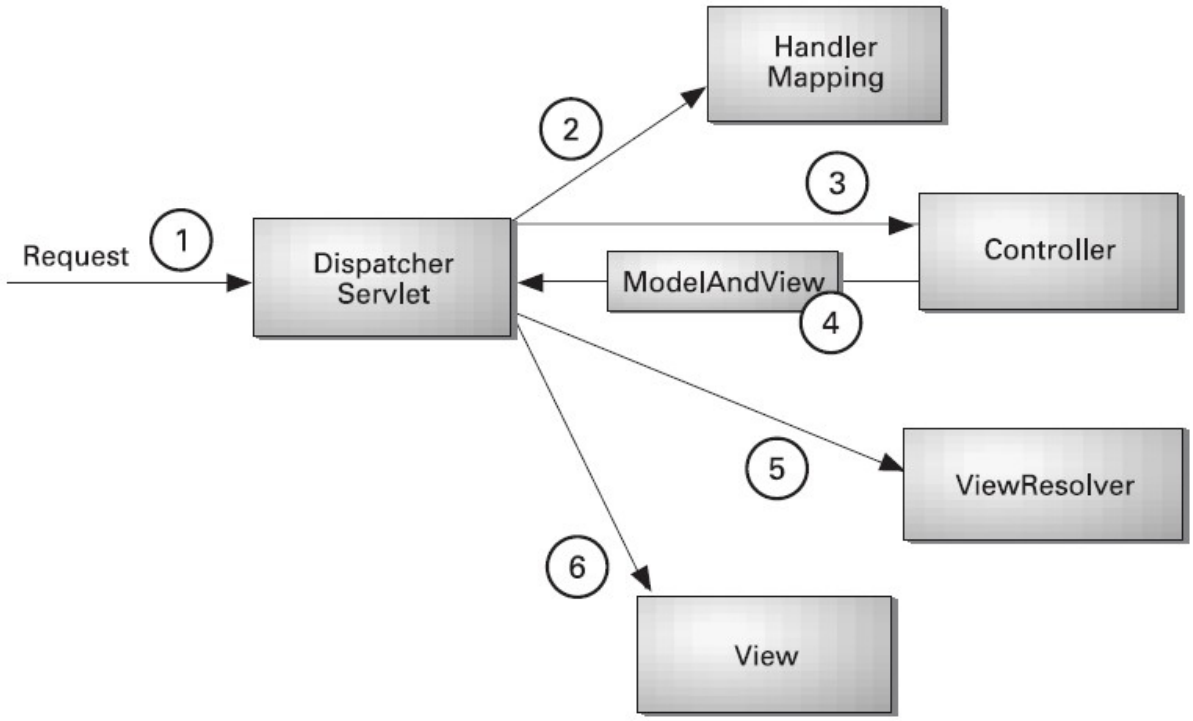
https://dzone.com/articles/spring-framework-restcontroller-vs-controller

Spring 4.0 introduced @RestController, a specialized version of the controller.

Adds @Controller and @ResponseBody by default.

Is the preferred method for creating mvc RESTful web services.





The Spring Web model-view-controller (MVC) framework is designed around a **DispatcherServlet** that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for uploading files. The default handler is based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods. With the introduction of Spring 3.0, the @Controller mechanism also allows you to create RESTful Web sites and applications, through the @PathVariable annotation and other features.

In previous versions of Spring, users were required to define one or more **HandlerMapping** beans in the web application context to map incoming web requests to appropriate handlers. With the introduction of annotated controllers, you generally don't need to do that because the RequestMappingHandlerMapping automatically looks for @RequestMapping annotations on all @Controller beans.

A **Controller** is typically responsible for preparing a model Map with data and selecting a view name but it can also write directly to the response stream and complete the request.

The **ViewResolver** provides a mapping between view names and actual views.

Por lo tanto, esto:

```
@RestController
public class HelloController {...}
```

Es lo mismo que esto:

```
@Controller
@ResponseBody
public class HelloController {...}
```

The RESTful web service controller simply returns the object and the object data is written directly to the http response as json/xml.

@ResponseBody

```
@Target(value={TYPE,METHOD})
 @Retention(value=RUNTIME)
 @Documented
public @interface ResponseBody
```

Annotation that indicates a method return value should be bound to the web response body. Supported for annotated handler methods.
As of version 4.0 this annotation can also be added on the type level in which case it is inherited and does not need to be added on the method level.

The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON or XML and passed back into the HttpResponse object.

@RequestMapping

```
@Target(value={TYPE,METHOD})
 @Retention(value=RUNTIME)
 @Documented
public @interface RequestMapping
```

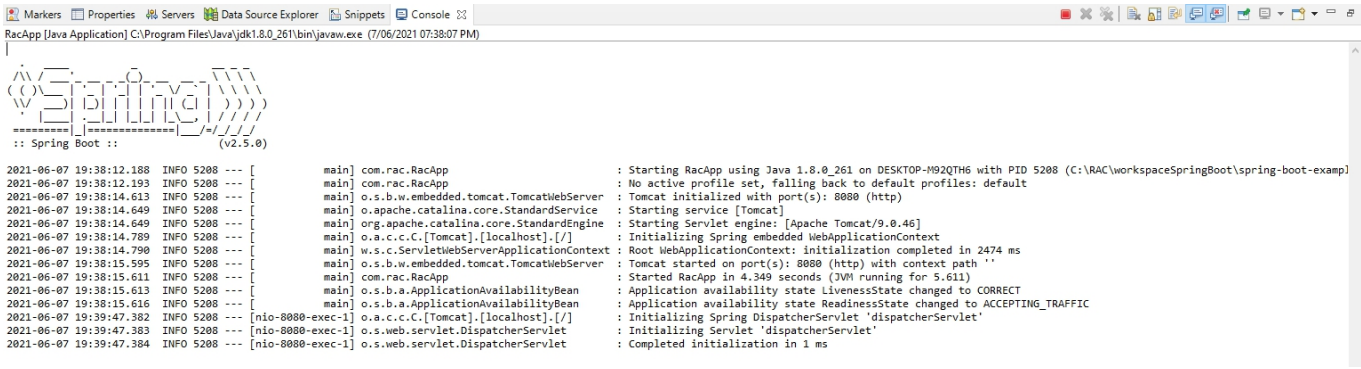
Annotation for mapping web requests onto methods in request-handling classes with flexible method signatures.

Annotation for mapping web requests onto methods in request-handling classes with flexible method signatures.
Both Spring MVC and Spring WebFlux support this annotation through a RequestMappingHandlerMapping and RequestMappingHandlerAdapter in their respective modules and package structure. For the exact list of supported handler method arguments and return types in each, please use the reference documentation links below:

Spring MVC Method Arguments and Return Values
Spring WebFlux Method Arguments and Return Values
Note: This annotation can be used both at the class and at the method level. In most cases, at the method level applications will prefer to use one of the HTTP method specific variants @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, or @PatchMapping.

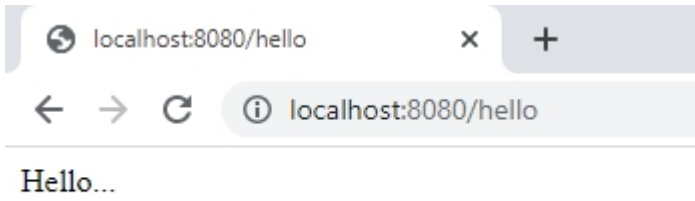
NOTE: When using controller interfaces (e.g. for AOP proxying), make sure to consistently put all your mapping annotations - such as @RequestMapping and @SessionAttributes - on the controller interface rather than on the implementation class.

Cuando se inicia la aplicación...



```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console
RacApp [Java Application] C:\Program Files\Java\jdk1.8.0_261\bin\javaw.exe (7/06/2021 07:38:07 PM)

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.5.0)

2021-06-07 19:38:12.188  INFO 5208 --- [           main] com.rac.RacApp                           : Starting RacApp using Java 1.8.0_261 on DESKTOP-M92QTH6 with PID 5208 (C:\RAC\workspaceSpringBoot\spring-boot-exampl
2021-06-07 19:38:12.193  INFO 5208 --- [           main] com.rac.RacApp                           : No active profile set, falling back to default profiles: default
2021-06-07 19:38:14.613  INFO 5208 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-06-07 19:38:14.649  INFO 5208 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-06-07 19:38:14.649  INFO 5208 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.46]
2021-06-07 19:38:14.789  INFO 5208 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-06-07 19:38:14.790  INFO 5208 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2474 ms
2021-06-07 19:38:15.595  INFO 5208 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-06-07 19:38:15.611  INFO 5208 --- [           main] com.rac.RacApp                           : Started RacApp in 4.349 seconds (JVM running for 5.611)
2021-06-07 19:38:15.613  INFO 5208 --- [           main] o.s.b.a.ApplicationAvailabilityBean      : Application availability state LivenessState changed to CORRECT
2021-06-07 19:38:15.616  INFO 5208 --- [           main] o.s.b.a.ApplicationAvailabilityBean      : Application availability state ReadinessState changed to ACCEPTING_TRAFFIC
2021-06-07 19:39:47.382  INFO 5208 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-06-07 19:39:47.383  INFO 5208 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2021-06-07 19:39:47.384  INFO 5208 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
```

Y se ingresa a la URL: http://localhost:8080/hello

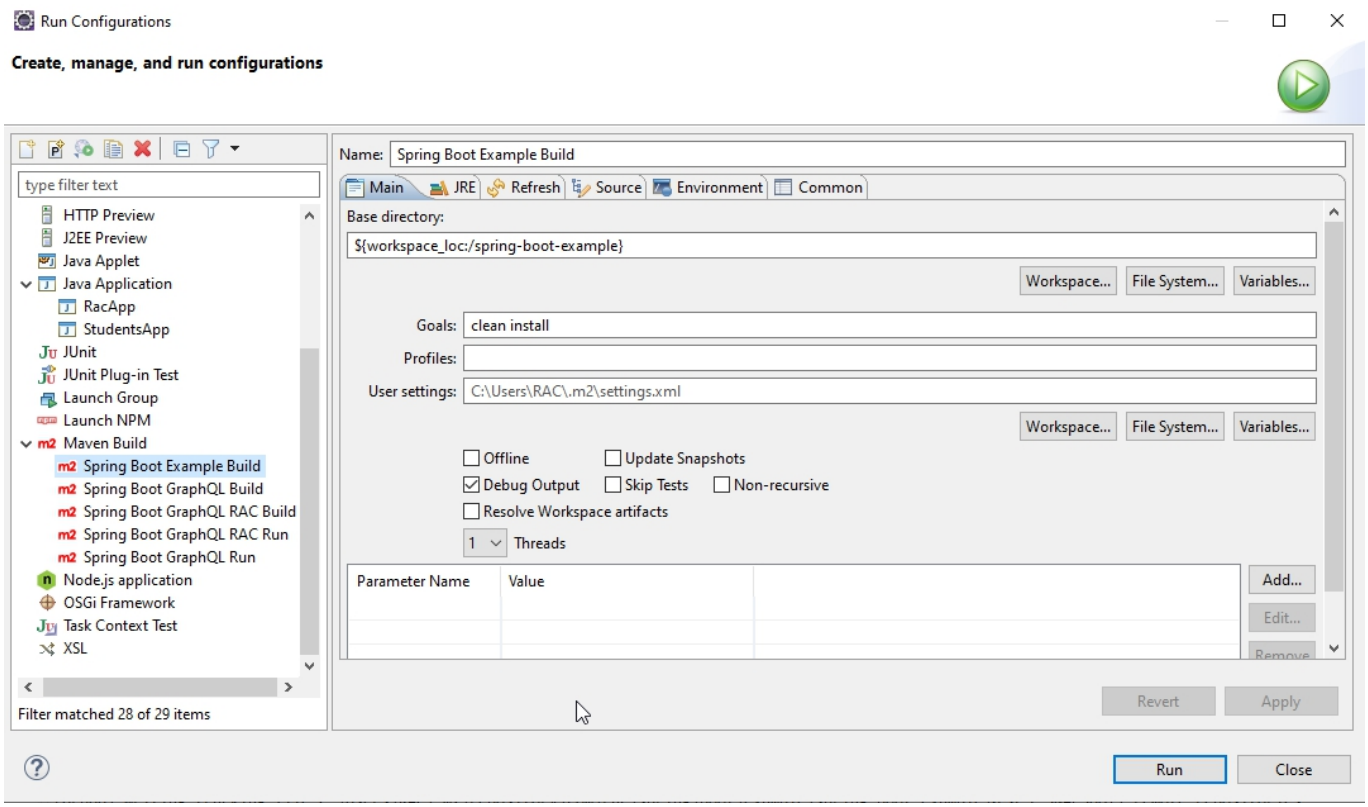


Crear un JAR Ejecutable

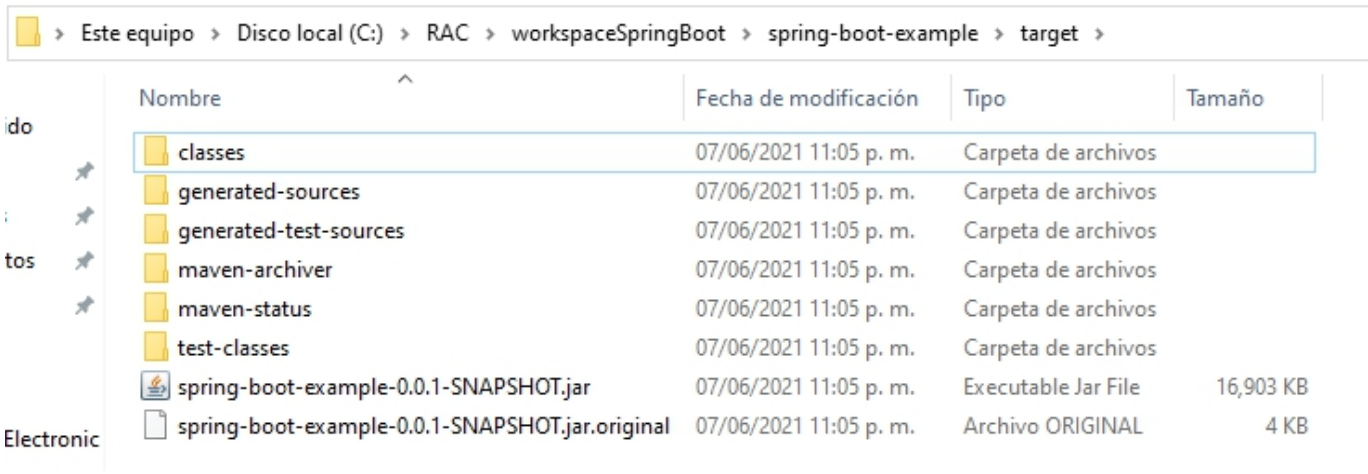
To create an executable jar, we need to add the spring-boot-maven-plugin to our pom.xml. To do so, insert the following lines just below the dependencies section:

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

Y creamos un Run Configuration en Eclipse:



Después de correr la configuración anterior obtenemos los siguiente:



Podemos ver como se creó el archivo:

- **spring-boot-example-0.0.1-SNAPSHOT.jar**

@Service

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Component
public @interface Service
```

Indicates that an annotated class is a "Service", originally defined by Domain-Driven Design (Evans, 2003) as "an operation offered as an interface that stands alone in the model, with no encapsulated state."

May also indicate that a class is a "Business Service Facade" (in the Core J2EE patterns sense), or something similar. This annotation is a general-purpose stereotype and individual teams may narrow their semantics and use as appropriate.

This annotation serves as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.

We mark beans with @Service to indicate that they're holding the business logic. Besides being used in the service layer, there isn't any other special use for this annotation.

Thing REST Endpoint

Vamos a crear un modelo llamado Thing, un controller llamado ThingController, una clase ThingService y una clase ThingData.

```
package rafael.alcocer.caldera.model;

public class Thing {

    private int id;
    private String name;
    private String description;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

```java
package rafael.alcocer.caldera.data;

import org.springframework.stereotype.Component;

import rafael.alcocer.caldera.model.Thing;

@Component
public class ThingData {

    public Thing generateThingData() {
        Thing thing = new Thing();
        thing.setId(13);
        thing.setName("NEW THING NAME");
        thing.setDescription("is the pattern of narrative development that aims to make vivid a
place, object, character, or group");

        return thing;
    }
}
```

```java
package rafael.alcocer.caldera.service;

import org.springframework.stereotype.Service;

import rafael.alcocer.caldera.data.ThingData;
import rafael.alcocer.caldera.model.Thing;

@Service
public class ThingService {

    private final ThingData thingData;

    public ThingService(ThingData thingData) {
        this.thingData = thingData;
    }

    public Thing generateThingData() {
        return thingData.generateThingData();
    }
}
```

```java
package rafael.alcocer.caldera.controller;

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

import rafael.alcocer.caldera.model.Thing;
import rafael.alcocer.caldera.service.ThingService;

@RestController
public class ThingController {

    private final ThingService thingService;

    public ThingController(ThingService thingService) {
        this.thingService = thingService;
    }

    @RequestMapping("/thing")
    public Thing showThing() {
        return thingService.generateThingData();
    }
}
```