## Maven—Spring-boot-starter

**Maven users can inherit from the spring-boot-starter-parent project to obtain sensible defaults. The parent project provides the following features:**

* **Java 1.8 as the default compiler level.**
* **UTF-8 source encoding.**
* **A**[**Dependency Management section**](https://docs.spring.io/spring-boot/docs/2.1.5.RELEASE/reference/htmlsingle/#using-boot-dependency-management)**, inherited from the spring-boot-dependencies pom, that manages the versions of common dependencies. This dependency management lets you omit <version> tags for those dependencies when used in your own pom.**
* **An execution of the**[**repackage goal**](https://docs.spring.io/spring-boot/docs/2.1.5.RELEASE/maven-plugin/repackage-mojo.html)**with a repackage execution id.**
* **Sensible**[**resource filtering**](https://maven.apache.org/plugins/maven-resources-plugin/examples/filter.html)**.**
* **Sensible plugin configuration (**[**exec plugin**](https://www.mojohaus.org/exec-maven-plugin/)**, [Git commit ID](https://github.com/ktoso/maven-git-commit-id-plugin" \t "_top), and**[**shade**](https://maven.apache.org/plugins/maven-shade-plugin/)**).**
* **Sensible resource filtering for application.properties and application.yml including profile-specific files (for example, application-dev.properties andapplication-dev.yml)**

**Note that, since the application.properties and application.yml files accept Spring style placeholders (${…​}), the Maven filtering is changed to use @..@placeholders. (You can override that by setting a Maven property called resource.delimiter.)**

### **13.2.1 Inheriting the Starter Parent**

**To configure your project to inherit from the spring-boot-starter-parent, set the parent as follows:**

***<!-- Inherit defaults from Spring Boot -->***

**<parent>**

**<groupId>org.springframework.boot</groupId>**

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

**<version>2.1.5.RELEASE</version>**

**</parent>**

|  |
| --- |
| **[Note]** |
| **You should need to specify only the Spring Boot version number on this dependency. If you import additional starters, you can safely omit the version number.** |

**With that setup, you can also override individual dependencies by overriding a property in your own project. For instance, to upgrade to another Spring Data release train, you would add the following to your pom.xml:**

**<properties>**

**<spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version>**

**</properties>**

### **13.2.2 Using Spring Boot without the Parent POM**

**Not everyone likes inheriting from the spring-boot-starter-parent POM. You may have your own corporate standard parent that you need to use or you may prefer to explicitly declare all your Maven configuration.**

**If you do not want to use the spring-boot-starter-parent, you can still keep the benefit of the dependency management (but not the plugin management) by using ascope=import dependency, as follows:**

**<dependencyManagement>**

**<dependencies>**

**<dependency>**

***<!-- Import dependency management from Spring Boot -->***

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-dependencies</artifactId>**

**<version>2.1.5.RELEASE</version>**

**<type>pom</type>**

**<scope>import</scope>**

**</dependency>**

**</dependencies>**

**</dependencyManagement>**

**The preceding sample setup does not let you override individual dependencies by using a property, as explained above. To achieve the same result, you need to add an entry in the dependencyManagement of your project before the spring-boot-dependencies entry. For instance, to upgrade to another Spring Data release train, you could add the following element to your pom.xml:**

**<dependencyManagement>**

**<dependencies>**

***<!-- Override Spring Data release train provided by Spring Boot -->***

**<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>2.1.5.RELEASE</version>**

**<type>pom</type>**

**<scope>import</scope>**

**</dependency>**

**</dependencies>**

**</dependencyManagement>**

|  |
| --- |
| **[Note]** |
| **In the preceding example, we specify a BOM, but any dependency type can be overridden in the same way.** |

### **13.2.3 Using the Spring Boot Maven Plugin**

**Spring Boot includes a**[**Maven plugin**](https://docs.spring.io/spring-boot/docs/2.1.5.RELEASE/reference/htmlsingle/#build-tool-plugins-maven-plugin)**that can package the project as an executable jar. Add the plugin to your <plugins> section if you want to use it, as shown in the following example:**

**<build>**

**<plugins>**

**<plugin>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-maven-plugin</artifactId>**

**</plugin>**

**</plugins>**

**</build>**

|  |
| --- |
| **[Note]** |
| **If you use the Spring Boot starter parent pom, you need to add only the plugin. There is no need to configure it unless you want to change the settings defined in the parent.** |

**Spring Boot @RestController**

**Spring Boot @RestController tutorial shows how to use the @RestController annotation in a Spring application to build a Restful controller.**

***Spring* is a popular Java application framework and *Spring Boot* is an evolution of Spring that helps create stand-alone, production-grade Spring based applications easily.**

**Spring MVC**

***Spring MVC* is the primary web framework built on the Servlet API. It is build on the popular MVC design pattern. *MVC (Model-View-Controller)* is a software architecture pattern, which separates application into three areas: model, view, and controller. The model represents a Java object carrying data. The view represents the visualization of the data that the model contains. The controller controls the data flow into model object and updates the view when the data changes. It separates the view and model.**

**Spring Framework 5.0 introduced a parallel reactive stack web framework called *Spring WebFlux*.**

**@RestController**

**@RestController is a convenience annotation for creating Restful controllers. It is a specialization of @Component and is autodetected through classpath scanning. It adds the @Controller and @ResponseBody annotations. It converts the response to JSON or XML. It does not work with the view technology, so the methods cannot return ModelAndView. It is typically used in combination with annotated handler methods based on the @RequestMapping annotation.**

**The @Controller annotation is used with the view technology.**

**Restful application**

**A *RESTFul application* follows the REST architectural style, which is used for designing networked applications. RESTful applications generate HTTP requests performing CRUD (Create/Read/Update/Delete) operations on resources. RESTFul applications typically return data in JSON or XML format.**

**Spring Boot @RestController example**

**In the following application, we demonstrate the usage of @RestController. The application returns a list of cities as JSON data.**

**pom.xml**

**src**

**├───main**

**│ ├───java**

**│ │ └───com**

**│ │ └───zetcode**

**│ │ │ Application.java**

**│ │ ├───controller**

**│ │ │ MyController.java**

**│ │ ├───model**

**│ │ │ City.java**

**│ │ └───service**

**│ │ CityService.java**

**│ │ ICityService.java**

**│ └───resources**

**│ └───static**

**│ index.html**

**└───test**

**└───java**

**This is the project structure.**

**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**

**http://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<groupId>com.zetcode</groupId>**

**<artifactId>springbootrestcontrollerex</artifactId>**

**<version>1.0-SNAPSHOT</version>**

**<packaging>jar</packaging>**

**<properties>**

**<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>**

**<maven.compiler.source>11</maven.compiler.source>**

**<maven.compiler.target>11</maven.compiler.target>**

**</properties>**

**<parent>**

**<groupId>org.springframework.boot</groupId>**

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

**<version>2.1.1.RELEASE</version>**

**</parent>**

**<dependencies>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

**</dependencies>**

**<build>**

**<plugins>**

**<plugin>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-maven-plugin</artifactId>**

**</plugin>**

**</plugins>**

**</build>**

**</project>**

**This is the Maven pom.xml file. The spring-boot-starter-parent is a parent POM providing dependency and plugin management for applications built with Maven. The spring-boot-starter-web is a starter for building web, including RESTful, applications using Spring MVC. It uses Tomcat as the default embedded container. The spring-boot-maven-plugin packages Spring applications into executable JAR or WAR archives.**

**com/zetcode/model/City.java**

**package com.zetcode.model;**

**import java.util.Objects;**

**public class City {**

**private Long id;**

**private String name;**

**private int population;**

**public City() {**

**}**

**public City(Long id, String name, int population) {**

**this.id = id;**

**this.name = name;**

**this.population = population;**

**}**

**public Long getId() {**

**return id;**

**}**

**public void setId(Long id) {**

**this.id = id;**

**}**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public int getPopulation() {**

**return population;**

**}**

**public void setPopulation(int population) {**

**this.population = population;**

**}**

**@Override**

**public boolean equals(Object o) {**

**if (this == o) return true;**

**if (o == null || getClass() != o.getClass()) return false;**

**City city = (City) o;**

**return population == city.population &&**

**Objects.equals(id, city.id) &&**

**Objects.equals(name, city.name);**

**}**

**@Override**

**public int hashCode() {**

**return Objects.hash(id, name, population);**

**}**

**@Override**

**public String toString() {**

**final StringBuilder sb = new StringBuilder("City{");**

**sb.append("id=").append(id);**

**sb.append(", name='").append(name).append('\'');**

**sb.append(", population=").append(population);**

**sb.append('}');**

**return sb.toString();**

**}**

**}**

**This is a City bean. It has id, name, and population attributes.**

**com/zetcode/controller/MyController.java**

**package com.zetcode.controller;**

**import com.zetcode.model.City;**

**import com.zetcode.service.ICityService;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.web.bind.annotation.GetMapping;**

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

**import java.util.List;**

**@RestController**

**public class MyController {**

**@Autowired**

**private ICityService cityService;**

**@GetMapping(value = "/cities")**

**public List<City> getCities() {**

**List<City> cities = cityService.findAll();**

**return cities;**

**}**

**}**

**This is MyController. It returns a list of cities in JSON format.**

**@RestController**

**public class MyController {**

**MyController is annotated with the @RestController annotation.**

**@Autowired**

**private ICityService cityService;**

**We inject the CityService with the @Autowired annotation.**

**@GetMapping(value = "/cities")**

**public List<City> getCities() {**

**The getCities() method is mapped to the getCities URL pattern; it returns a list of cities, which is converted to JSON by a message converter.**

**com/zetcode/service/ICityService.java**

**package com.zetcode.service;**

**import com.zetcode.bean.City;**

**import java.util.List;**

**public interface ICityService {**

**public List<City> findAll();**

**}**

**The ICityService contains the findAll() contract method.**

**com/zetcode/service/CityService.java**

**package com.zetcode.service;**

**import com.zetcode.model.City;**

**import java.util.ArrayList;**

**import java.util.List;**

**import org.springframework.stereotype.Service;**

**@Service**

**public class CityService implements ICityService {**

**@Override**

**public List<City> findAll() {**

**var cities = new ArrayList<City>();**

**cities.add(new City(1L, "Bratislava", 432000));**

**cities.add(new City(2L, "Budapest", 1759000));**

**cities.add(new City(3L, "Prague", 1280000));**

**cities.add(new City(4L, "Warsaw", 1748000));**

**cities.add(new City(5L, "Los Angeles", 3971000));**

**cities.add(new City(6L, "New York", 8550000));**

**cities.add(new City(7L, "Edinburgh", 464000));**

**cities.add(new City(8L, "Berlin", 3671000));**

**return cities;**

**}**

**}**

**The CityService contains the implementation of the findAll() method.**

**resources/static/index.html**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Home page</title>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**</head>**

**<body>**

**<p>**

**<a href="cities">Get all cities</a>**

**</p>**

**</body>**

**</html>**

**This is the home page. It contains a link to get all cities.**

**com/zetcode/Application.java**

**package com.zetcode;**

**import org.springframework.boot.SpringApplication;**

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

**@SpringBootApplication**

**public class Application {**

**public static void main(String[] args) {**

**SpringApplication.run(Application.class, args);**

**}**

**}**

**Application is the entry point which sets up Spring Boot application. The @SpringBootApplication annotation enables auto-configuration and component scanning. During the scanning process, the @RestController annotation is looked up and a Spring bean is created from the MyController class.**

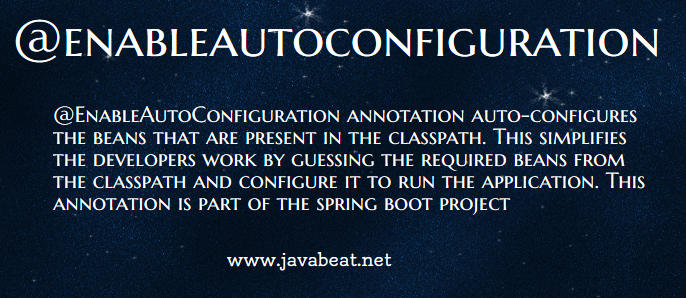
**$ mvn spring-boot:run**

**After the application is run, we can navigate to localhost:8080.**

# @EnableAutoConfiguration Annotation in Spring Boot

**@EnableAutoConfiguration annotation auto-configures the beans that are present in the classpath. This simplifies the developers work by guessing the required beans from the classpath and configure it to run the application. This**[**annotation**](https://javabeat.net/annotations-in-java-5-0/)**is part of the**[**spring boot**](https://javabeat.net/spring-boot/)**project.**

**For example, if you have tomcat-embedded.jar in the classpath, then you will need a TomcatEmbeddedServletContainerFactory bean to configure the tomcat server. This will be searched and configured without any manual XML configurations.**

**Spring Boot Enable Auto Configuration**

**With the spring boot 1.2.0 release, the need for this annotation has been reduced because there is an alternative annotation**[**@SpringBootApplication**](https://javabeat.net/springbootapplication-annotation/)**which combines the three annotations**[**@Configuration**](https://javabeat.net/javaconfig-spring-3-0/)**,@EnableAutoConfiguration and code>**[**@ComponentScan.**](https://javabeat.net/spring-mvc-component-scan-annotations/)

**The package of the class that is annotated with @EnableAutoConfiguration has specific significance and is often used as a ‘default’. For example, it will be used when scanning for**[**@Entity**](https://javabeat.net/spring-data-jpa-namedquery/)**classes. It is generally recommended that you place @EnableAutoConfiguration in a root package so that all sub-packages and classes can be searched.**

**Auto-configuration classes are normal @Configuration annotated classes only. These are mentioned in the spring.factories file. Spring checks the spring.factories files under the folder META-INF in your project or JAR file to auto-configure the configuration classes.**

## @EnableAutoConfiguration Parameters

**The following are the parameters that can be passed inside this annotation:**

* **exclude – Exclude the list of classes from the auto configuration.**
* **excludeNames – Exclude the list of fully qualified class names from the auto configuration. This parameter added since spring boot 1.3.0.**

**The above parameters helps you to exclude the list of configuration classes that are not required to be auto-configured.**

**Here is the sample snippet for how to use the parameters:**

**[](https://is.ltroute.com/click.track?CID=420780&AFID=426982&SID=javabeat&nonencodedurl=https://101electronicsstore.com/products/cable-organizer-travel-bag)**

**@EnableAutoConfiguration(exclude={Book.class})**

## Write Custom Auto-Configuration

**I have written a very simple auto-configuration module for spring boot application. Note that you can write auto-configuration as part of the current application or you can import the JAR file. The important point is to add the spring.factories file under the META-INF folder. This is the default behavior of**[**spring application**](https://javabeat.net/spring-boot-spring-mvc/)**to search for that file.**

**Here is the steps to write your own auto-configuration class:**

**1.  Create a @Configuration class as one below:**

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;**

**import org.springframework.context.annotation.Bean;**

**import org.springframework.context.annotation.Configuration;**

**@Configuration**

**@ConditionalOnClass({ String.class })**

**public class ConfigureDefaults {**

**Logger logger = LoggerFactory.getLogger(ConfigureDefaults.class);**

**@Bean**

**public String cacheManager() {**

**logger.info("Configure Defaults");**

**return new String("test");**

**}**

**}**

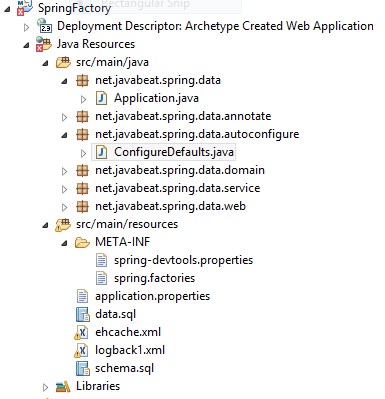
**2. Create spring.factories file as below and put it under the META-INF folder:**

**org.springframework.boot.autoconfigure.EnableAutoConfiguration=\**

**net.javabeat.spring.data.autoconfigure.ConfigureDefaults**

**3. Run your spring boot application.**

**You can notice that ConfigureDefaults will be invoked and the beans defined in that classes will be configured for the application use. Look at the below project structure I have used for testing my**[**spring boot**](https://javabeat.net/spring-boot/)**application with the custom auto-configuration module.**

**Spring Boot Auto Configuration**

# Difference between @SpringBootApplication vs @EnableAutoConfiguration annotations in Spring Boot

**Even though both @SpringBootApplication and @EnableAutoConfiguration can be used to enable the auto-configuration feature of Spring Boot there is a subtle difference between them. The @SpringBootApplication does much more than what @EnableAutoConfiguration do. It's actually a combination of three annotations: @Configuration which is used in Java-based configuration on Spring framework, @ComponentScan to enable component scanning of components you write e.g.**[**@Controller**](http://javarevisited.blogspot.sg/2017/11/difference-between-component-service.html)**classes, and @EnableAutoConfgiuration itself, which is used to enable auto-configuration in**[**Spring Boot**](https://javarevisited.blogspot.sg/2018/02/top-5-spring-microservices-courses-with-spring-boot-and-spring-cloud.html)**application. Spring Boot designers realize that these three annotations are frequently used together so they bundled them into @SpringBootApplicaiton. Now, instead of three annotations you just need to specify one annotation on your Main class.** **What is the difference between @EnableAutoConfiguration and @SpringBootApplication is also a common Spring Boot interview question and it was recently asked one of my friends on his Java interview. Given the popularity of**[**Spring Boot framework**](http://www.java67.com/2017/11/top-5-free-core-spring-mvc-courses-learn-online.html)**, it's good to know such questions before you go for your next Java interview.  
  
In this article, I'll explain what does the @SpringBootApplication and @EnableAutoConfiguration do and highlight some important difference between them so that you can answer this question with confidence if you happen to see them in your interview.**

## 1. What is Auto-Configuration in Spring Boot?

**If you don't know, Spring Boot aims to simplify Java development with Spring framework. Currently, Spring does a lot for you but in return, it also asks a lot from you in terms of configuration and dependency management, Spring Boot aims to solve that problem.  
  
The Spring Boto auto-configuration feature tries to automatically configure your Spring application based upon the JAR dependency you have added in the classpath.  
  
For example, if HSQLDB is present on your classpath and you have not configured any database manually, Spring will auto-configure an in-memory database for you.  
  
By default, this auto-configuration feature is not enabled and you need to opt-in for it by adding the @EnableAutoConfiguration or @SpringBootApplicaiton annotations to one of your @Configuration classes, generally the Main class which is used to run your application.**

**2. Difference between @EnableAutoConfiguration and @SpringBootApplication**

**Now that we know what is @SpringBootApplication and @EnableAutoConfiguration annotation and what do they do, it's time to highlight some important difference between them.  
  
Here is a couple of worth-noting difference between @SpringBootApplication and @EnableAutoConfiguration annotations of**[**Spring Boot**](http://javarevisited.blogspot.sg/2018/01/how-to-learn-spring-core-spring-mvc-boot-security-framework.html#axzz55IgfKjy8)**:  
  
1. Availability  
The @SpringBootApplicaiton is relatively new than @EnableAutoConfiguration. It was introduced in Spring Boot 1.2 release while @EnableAutoConfiguation is present form the Spring Boot 1.0 release.  
  
  
2. Purpose  
The clear purpose of @EnableAutoConfiguration is to enable automatic configuration feature of Spring Boot application which automatically configures things if certain classes are present in Classpath e.g. it can configure Thymeleaf TemplateResolver and ViewResolver if Thymeleaf is present in the [classpath](http://www.java67.com/2012/08/what-is-path-and-classpath-in-java-difference.html" \t "_blank).  
  
On the other hand, @SpringBootApplication does three things, it allows you to run the Main class as a JAR with an embedded container. It enables Java configuration and it also enables Component Scanning.  
  
  
3. Uses  
It's not mandatory to put @SpringBootApplication to create a**[**Spring Boot**](https://javarevisited.blogspot.sg/2018/04/top-5-java-frameworks-to-learn-in-2018_27.html)**application, you can still use @Configuration and @EnableAutoConfiguration individually as shown in the example given in next point.  
  
  
4. Control  
The @EnableAutoConfiguration annotation allows you to selectively exclude certain classes from auto-configuration using exclude attribute as shown below:**

**@Configuration**

**@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})**

**public class MyConfiguration {**

**//.. Java code**

**}**

**If the class is not on the classpath, you can use the excludeName attribute of the @EnableAutoConfiguration annotation and specify the fully qualified class name.  
  
If you want to learn more about this annotation, I suggest you reading Craig Walls' (author of**[**Spring in Action**](http://www.java67.com/2016/12/5-spring-framework-books-for-java-programmers.html)**) another masterpiece**[**Spring Boot in Action**](https://www.amazon.com/Spring-Boot-Action-Craig-Walls/dp/1617292540?tag=javamysqlanta-20)**.**

**3. Important points**

**Now that you understand the difference between @EnableAutoConfiguration and @SpringBootApplication, it's time to revise some important points about these two important annotations.  
  
1. You should annotate the Main class or Bootstrap class with the @SpringBootApplication, this will allow you to run as a JAR with embedded web server**[**Tomcat**](https://javarevisited.blogspot.sg/2017/01/12-essential-apache-web-server-interview-questions-answers-java-linux.html)**. If you want you can change that to Jetty or Undertow.  
  
  
2. The @SpringBootApplication is a combination of three annotations @Configuration (used for Java-based configuration), @ComponentScan (used for component scanning), and @EnableAutoConfiguration (used to enable auto-configuration in**[**Spring Boot**](http://javarevisited.blogspot.sg/2018/01/how-to-learn-spring-core-spring-mvc-boot-security-framework.html)**).  
  
  
3. The @EnableAutoConfiguration annotations enable auto-configuration features of Spring Boot which configures modules based on the presence of certain classes on the classpath. For example, if Thymeleaf JAR is present in classpath and**[**Spring MVC**](https://javarevisited.blogspot.sg/2017/06/how-spring-mvc-framework-works-web-flow.html)**is enabled e.g. using spring-boot-web-starter package then it can automatically configure template resolver and view resolver for you.  
  
  
4. The @EnableAutoConfiguration annotation is based on @Conditional annotation of Spring 4.0 which enables conditional configuration.  
  
  
5. In case of auto-configuration, manually declared beans can override beans automatically created by auto-configuration feature. This is achieved by using @ConditionalOnMissingBean of Spring 4.0  
  
  
6. If you are using @EnableAutoConfiguration classes then you can selectively exclude certain classes from auto-configuration by using exclude as shown below:  
  
@EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)  
  
  
7. The @SpringBootApplication annotation also provides aliases to customize the attributes of @EnableAutoConfiguration and @ComponentScan annotations.  
  
  
That's all about the difference between @SpringBootApplication and @EnableAutoConfiguration annotations of Spring Boot. As you learned, the @SpringBootApplication makes it easy to enable auto-configuration and create a Bootstrap class by reducing the number of annotations you normally need i.e. instead of 3 annotations you just need one. But, if you need more control over autoconfiguration then you should use @EnableAutoConfiguration, which allows you to exclude classes from auto-configuration.**

# @RestController vs. @Controller

## Spring MVC Framework and REST

**Spring’s annotation-based MVC framework simplifies the process of creating RESTful web services. The key difference between a traditional Spring MVC controller and the RESTful web service controller is the way the HTTP response body is created. While the traditional MVC controller relies on the View technology, the RESTful web service controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML.  For a detailed description of creating RESTful web services using the Spring framework, click**[**here**](http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/mvc.html)**.**

### **Spring MVC REST Workflow**

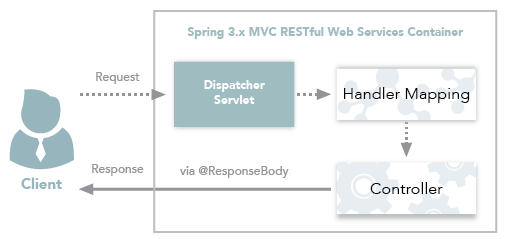
**The following steps describe a typical Spring MVC REST workflow:**

1. **The client sends a request to a web service in URI form.**
2. **The request is intercepted by the DispatcherServlet which looks for Handler Mappings and its type.**
   * **The Handler Mappings section defined in the application context file tells DispatcherServlet which strategy to use to find controllers based on the incoming request.**
   * **Spring MVC supports three different types of mapping request URIs to controllers: annotation, name conventions, and explicit mappings.**
3. **Requests are processed by the Controller and the response is returned to the DispatcherServlet which then dispatches to the view.**

**In Figure 1, notice that in the traditional workflow the ModelAndView object is forwarded from the controller to the client. Spring lets you return data directly from the controller, without looking for a view, using the @ResponseBody annotation on a method. Beginning with Version 4.0, this process is simplified even further with the introduction of the @RestController annotation. Each approach is explained below.**

## Using the @ResponseBody Annotation

**When you use the @ResponseBody annotation on a method, Spring converts the return value and writes it to the HTTP response automatically. Each method in the Controller class must be annotated with @ResponseBody.**

****

**Figure 2: Spring 3.x MVC RESTful web services workflow**

### **Behind the Scenes**

**Spring has a list of HttpMessageConverters registered in the background. The responsibility of the HTTPMessageConverter is to convert the request body to a specific class and back to the response body again, depending on a predefined mime type. Every time an issued request hits @ResponseBody, Spring loops through all registered HTTPMessageConverters seeking the first that fits the given mime type and class, and then uses it for the actual conversion.**

### **Code Example**

**Let’s walk through @ResponseBody with a simple example.**

#### **Project Creation and Setup**

1. **Create a Dynamic Web Project with Maven support in your Eclipse or [MyEclipse IDE.](https://www.genuitec.com/dow/products/myeclipse/" \t "_blank)**
2. **Configure Spring support for the project. If you are using Eclipse IDE, you need to download all Spring dependencies and configure your pom.xml to contain those dependencies. In MyEclipse, you only need to**[**install the Spring facet**](https://www.genuitec.com/products/myeclipse/learning-center/spring/myeclipse-spring-development-overview/)**and the rest of the configuration happens automatically.**
3. **Create the following Java class named Employee. This class is our POJO.**

**package com.example.spring.model;**

**import javax.xml.bind.annotation.XmlRootElement;**

**@XmlRootElement(name = "Employee")**

**public class Employee {**

**String name;**

**String email;**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public String getEmail() {**

**return email;**

**}**

**public void setEmail(String email) {**

**this.email = email;**

**}**

**public Employee() {**

**}**

**}**

**Then, create the following @Controller class:**

**package com.example.spring.rest;**

**import org.springframework.stereotype.Controller;**

**import org.springframework.web.bind.annotation.PathVariable;**

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

**import org.springframework.web.bind.annotation.RequestMethod;**

**import org.springframework.web.bind.annotation.ResponseBody;**

**import com.example.spring.model.Employee;**

**@Controller**

**@RequestMapping("employees")**

**public class EmployeeController {**

**Employee employee = new Employee();**

**@RequestMapping(value = "/{name}", method = RequestMethod.GET, produces = "application/json")**

**public @ResponseBody Employee getEmployeeInJSON(@PathVariable String name) {**

**employee.setName(name);**

**employee.setEmail("employee1@genuitec.com");**

**return employee;**

**}**

**@RequestMapping(value = "/{name}.xml", method = RequestMethod.GET, produces = "application/xml")**

**public @ResponseBody Employee getEmployeeInXML(@PathVariable String name) {**

**employee.setName(name);**

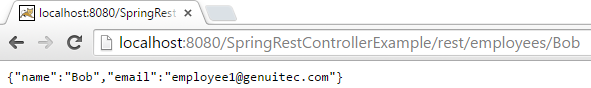
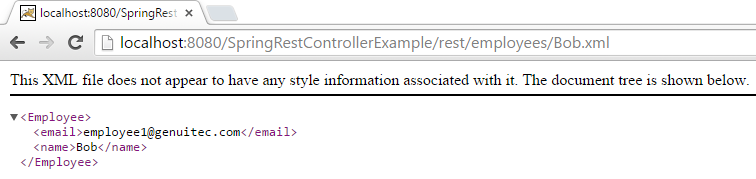
**employee.setEmail("employee1@genuitec.com");**

**return employee;**

**}**

**}**

**Notice the @ResponseBody added to each of the @RequestMapping methods in the return value. After that, it's a two-step process:**

1. **Add the <context:component-scan> and <mvc:annotation-driven /> tags to the Spring configuration file.**
   * **<context:component-scan> activates the annotations and scans the packages to find and register beans within the application context.**
   * **<mvc:annotation-driven/> adds support for reading and writing JSON/XML if the Jackson/JAXB libraries are on the classpath.**
   * **For JSON format, include the jackson-databind jar and for XML include the jaxb-api-osgi jar to the project classpath.**
2. **Deploy and run the application on any server (e.g., Tomcat). If you are using**[**MyEclipse**](https://www.genuitec.com/dow/products/myeclipse/)**, you can run the project on the**[**embedded Tomcat server**](https://www.genuitec.com/products/myeclipse/learning-center/deploy/myeclipse-tomcat-server/)**.  
     
   JSON—Use the URL: http://localhost:8080/SpringRestControllerExample/rest/employees/Bob and the following output displays:  
     
   XML — Use the   
   URL: http://localhost:8080/SpringRestControllerExample/rest/employees/Bob.xml and the following output displays:**

## Using the @RestController Annotation

**Spring 4.0 introduced @RestController, a specialized version of the controller which is a convenience annotation that does nothing more than add the @Controller and @ResponseBody annotations. By annotating the controller class with @RestController annotation, you no longer need to add @ResponseBody to all the request mapping methods. The @ResponseBody annotation is active by default. Click**[**here**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html)**to learn more.  
**

**To use @RestController in our example, all we need to do is modify the @Controller to @RestController and remove the @ResponseBody from each method. The resultant class should look like the following:**

**package com.example.spring.rest;**

**import org.springframework.web.bind.annotation.PathVariable;**

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

**import org.springframework.web.bind.annotation.RequestMethod;**

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

**import com.example.spring.model.Employee;**

**@RestController**

**@RequestMapping("employees")**

**public class EmployeeController {**

**Employee employee = new Employee();**

**@RequestMapping(value = "/{name}", method = RequestMethod.GET, produces = "application/json")**

**public Employee getEmployeeInJSON(@PathVariable String name) {**

**employee.setName(name);**

**employee.setEmail("employee1@genuitec.com");**

**return employee;**

**}**

**@RequestMapping(value = "/{name}.xml", method = RequestMethod.GET, produces = "application/xml")**

**public Employee getEmployeeInXML(@PathVariable String name) {**

**employee.setName(name);**

**employee.setEmail("employee1@genuitec.com");**

**return employee;**

**}**

**}**

**Note that we no longer need to add the @ResponseBody to the request mapping methods. After making the changes, running the application on the server again results in the same output as before.**

## Conclusion

**As you can see, using @RestController is quite simple and is the preferred method for creating MVC RESTful web services starting from Spring v4.0. I would like to extend a big thank you to my co-author, Swapna Sagi, for all of her help in bringing you this information!**