# [Spring Framework: @RestController(](https://www.genuitec.com/spring-frameworkrestcontroller-vs-controller/" \o "Permalink to Spring Framework: @RestController vs @Controller)*[Spring 4.x](https://www.genuitec.com/spring-frameworkrestcontroller-vs-controller/" \o "Permalink to Spring Framework: @RestController vs @Controller)*[) vs @Controller](https://www.genuitec.com/spring-frameworkrestcontroller-vs-controller/" \o "Permalink to Spring Framework: @RestController vs @Controller)(*Spring 3.x*)

**Spring MVC - @RequestBody and @ResponseBody (used before 4.x) demystified**

In this post i want to dig into spring mvc a little, revealing what happens behind the scenes when a request is converted to your parameter object and vice versa. Before we start, i want to explain the purpose of these annotations.

**What are @RequestBody and @ResponseBody for?**

They are annotations of the spring mvc framework and can be used in a controller to implement smart object serialization and deserialization. They help you avoid boilerplate code by extracting the logic of messageconversion and making it an aspect. Other than that they help you support multiple formats for a single REST resource without duplication of code. If you annotate a method with @ResponseBody, spring will try to convert its return value and write it to the http response automatically. If you annotate a methods parameter with @RequestBody, spring will try to convert the content of the incoming request body to your parameter object on the fly.  
  
Here is an example

**@Controller**

**@RequestMapping**(value = "/bookcase")

**public** **class** **BookCaseController** {

**private** BookCase bookCase;

**@RequestMapping**(method = RequestMethod.GET)

**@ResponseBody**

**public** BookCase **getBookCase**() {

**return** **this**.bookCase;

}

**@RequestMapping**(method = RequestMethod.PUT)

**@ResponseStatus**(HttpStatus.NO\_CONTENT)

**public** **void** **setBookCase**(**@RequestBody** BookCase bookCase) {

**this**.bookCase = bookCase;

}

}

**So what is Spring doing behind the scenes when we are using those Annotations?**

Depending on your configuration, spring has a list of [HttpMessageConverters](http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/http/converter/HttpMessageConverter.html) registered in the background. A HttpMessageConverters responsibility is to convert the request body to a specific class and back to the response body again, depending on a predefined [mime type](http://de.wikipedia.org/wiki/Multipurpose_Internet_Mail_Extensions). Every time an issued request is hitting a @RequestBody or @ResponseBody annotation spring loops through all registered HttpMessageConverters seeking for the first that fits the given mime type and class and then uses it for the actual conversion.

**How can i add a custom HttpMessageConverter?**

By adding @EnableWebMvc respectively <mvc:annotation-driven />, spring registers a bunch of predefined messageconverters for JSON/XML and so on. You can add a custom converter like the following

**@Configuration**

**@EnableWebMvc**

**@ComponentScan**

**public** **class** **WebConfiguration** **extends** WebMvcConfigurerAdapter {

**@Override**

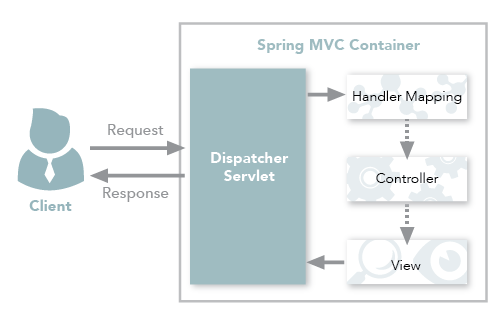
**public** **void** **configureMessageConverters**(List<HttpMessageConverter<?>> httpMessageConverters) {

httpMessageConverters.add(**new** BookCaseMessageConverter(**new** MediaType("text", "csv")));

}

**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).

  
*Figure 1: Spring MVC traditional workflow*

**Spring MVC REST Workflow**

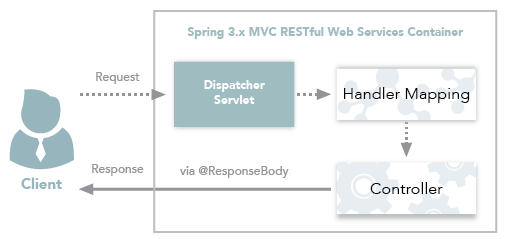
The following steps describe a typical Spring MVC REST workflow:

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.

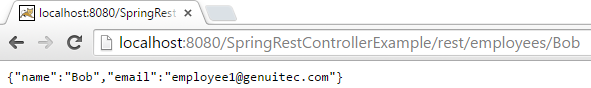
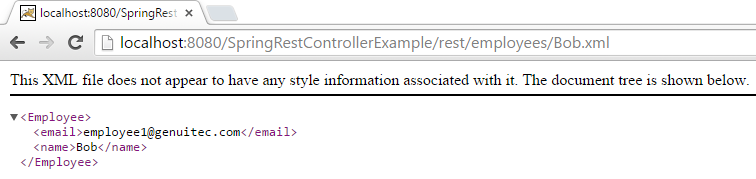
**Project Creation and Setup**

1. Create a Dynamic Web Project with Maven support in your Eclipse or MyEclipse IDE.
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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | 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() {      }    } |

1. Create the following @Controller class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | 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;        }    } |

1. Notice the @ResponseBody added to each of the @RequestMapping methods in the return value.
2. 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.
3. Deploy and run the application on any server (e.g., Tomcat). If you are using 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:  
   *Figure 3: JSON output***XML**—Use the URL: http://localhost:8080/SpringRestControllerExample/rest/employees/Bob.xml and the following output displays:  
   *Figure 4: XML output*

**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.  
  
*Figure 5: Spring 4.x MVC RESTful Web Services Workflow*  
  
@Target(value=TYPE)

@Retention(value=RUNTIME)

@Documented

[@Controller](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Controller.html)

[@ResponseBody](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html)

public @interface **RestController**

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | 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 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!

