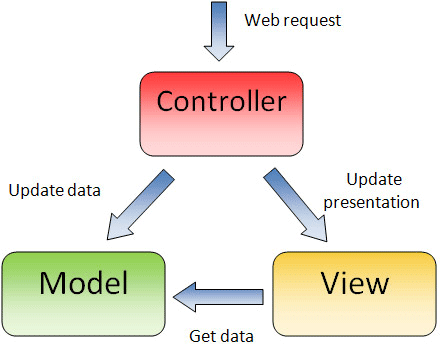
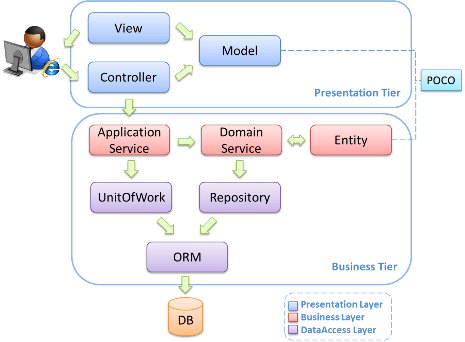
# **Spring MVC**

[Model-view-controller (MVC)](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller) is a well known [design pattern](https://howtodoinjava.com/category/design-patterns/) for designing UI based applications. It mainly decouples business logic from UIs by separating the roles of model, view, and controller in an application. Usually, models are responsible for encapsulating application data for views to present. Views should only present this data, without including any business logic. And controllers are responsible for receiving requests from users and invoking back-end services (manager or dao) for business logic processing. After processing, back-end services may return some data for views to present. Controllers collect this data and prepare models for views to present. The core idea of the MVC pattern is to separate business logic from UIs to allow them to change independently without affecting each other.



In a Spring MVC application, models usually consist of POJO objects that are processed by the service layer and persisted by the persistence layer. Views are usually JSP templates written with [Java Standard Tag Library (JSTL)](https://jstl.java.net/). Controller part is played by dispatcher servlet which we will learn about in this tutorial in more detail.

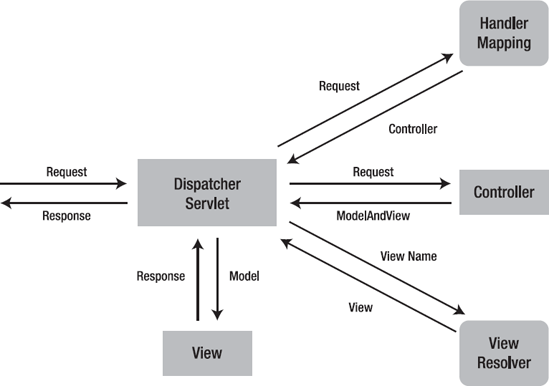
Some developers consider the service layer and DAO layers classes as part of model component in MVC. I have a different opinion on this. I do not consider service and DAO layers classes the part of MVC framework. Usually a web application is 3-tier architecture i.e. data-service-presentation. MVC is actually part of presentation layer.



**Dispatcher Servlet (Spring Controller)**

In the simplest Spring MVC application, a controller is the only servlet you need to configure in a Java web deployment descriptor (i.e., the web.xml file). A Spring MVC controller—often referred to as a [Dispatcher Servlet](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) implements [front controller](https://en.wikipedia.org/wiki/Front_Controller_pattern) design pattern and every web request must go through it so that it can manage the entire request life cycle.

When a web request is sent to a Spring MVC application, dispatcher servlet first receives the request. Then it organizes the different components configured in Spring’s web application context (e.g. actual request handler controller and view resolvers) or annotations present in the controller itself, all needed to handle the request.



To define a controller class in Spring 3.0, a class has to be marked with the [@Controller](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Controller.html) annotation. When a @Controller annotated controller receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller class map each request to a handler method by one or more handler mappings. In order to do so, a controller class’s methods are decorated with the [@RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) annotation, making them handler methods.

After a handler method has finished processing the request, it delegates control to a view, which is represented as handler method’s return value. To provide a flexible approach, a handler method’s return value doesn’t represent a view’s implementation but rather a logical view i.e. without any file extension. You can map these logical views to right implementation into applicationContext file so that you can easily change your view layer code without even touching request handler class code.

To resolve the correct file for a logical name is the responsibility of [view resolvers](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ViewResolver.html). Once the controller class has resolved a view name into a view implementation, per the view implementation design, it renders the objects.

**Handlers**

**Views and resolving them (View Resolver)**

All MVC frameworks for web applications provide a way to address views. Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology. Out of the box, Spring enables you to use JSPs, Velocity templates and XSLT views, for example.

The two interfaces which are important to the way Spring handles views are ***ViewResolver*** and ***View***. The ViewResolver provides a mapping between view names and actual views. The View interface addresses the preparation of the request and hands the request over to one of the view technologies.

### **Resolving views - the ViewResolver interface**

All controllers in the Spring Web MVC framework return a ModelAndView instance. Views in Spring are addressed by a view name and are resolved by a view resolver. Spring comes with quite a few view resolvers.

**View resolvers**

|  |  |
| --- | --- |
| **ViewResolver** | **Description** |
| AbstractCachingViewResolver | An abstract view resolver which takes care of caching views. Often views need preparation before they can be used, extending this view resolver provides caching of views. |
| XmlViewResolver | An implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is /WEB-INF/views.xml. |
| ResourceBundleViewResolver | An implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath. The default file name is views.properties. |
| UrlBasedViewResolver | A simple implementation of the ViewResolver interface that effects the direct resolution of symbolic view names to URLs, without an explicit mapping definition. This is appropriate if your symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings. |
| InternalResourceViewResolver | A convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses such as JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass(..). See the Javadocs for the UrlBasedViewResolver class for details. |
| VelocityViewResolver /FreeMarkerViewResolver | A convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) or FreeMarkerView respectively and custom subclasses of them. |

**InternalResourceViewResolver**

The **InternalResourceViewResolver** is used to resolve the provided URI to actual URI. The following example shows how to use the InternalResourceViewResolver using the Spring Web MVC Framework. The InternalResourceViewResolver allows mapping webpages with requests.

package com.tutorialspoint;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
import org.springframework.ui.ModelMap;  
  
@Controller  
@RequestMapping("/hello")  
public class HelloController{  
   
 @RequestMapping(method = RequestMethod.GET)  
 public String printHello(ModelMap model) {  
 model.addAttribute("message", "Hello Spring MVC Framework!");  
  
 return "hello";  
 }  
}

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name = "prefix" value = "/WEB-INF/jsp/"/>  
 <property name = "suffix" value = ".jsp"/>  
</bean>

For example, using the above configuration, if URI

* /hello is requested, DispatcherServlet will forward the request to the prefix + viewname + suffix = /WEB-INF/jsp/hello.jsp.

**XmlViewResolver**

The XmlViewResolver is used to resolve the view names using view beans defined in xml file. The following example shows how to use the XmlViewResolver using Spring Web MVC framework.

## **TestWeb-servlet.xml**

<bean class = "org.springframework.web.servlet.view.XmlViewResolver">  
 <property name = "location">  
 <value>/WEB-INF/views.xml</value>  
 </property>  
</bean>

## **views.xml**

<bean id = "hello"  
 class = "org.springframework.web.servlet.view.JstlView">  
 <property name = "url" value = "/WEB-INF/jsp/hello.jsp" />  
</bean>

For example, using the above configuration, if URI −

* /hello is requested, DispatcherServlet will forward the request to the hello.jsp defined by bean hello in the view.xml.

**ResourceBundleViewResolver**

The ResourceBundleViewResolver is used to resolve the view names using view beans defined in the properties file. The following example shows how to use the ResourceBundleViewResolver using the Spring Web MVC Framework.

## **TestWeb-servlet.xml**

<bean class = "org.springframework.web.servlet.view.ResourceBundleViewResolver">  
 <property name = "basename" value = "views" />  
</bean>

Here, the basename refers to name of the resource bundle, which carries the views. The default name of the resource bundle is views.properties, which can be overridden using the basename property.

## **views.properties**

hello.(class) = org.springframework.web.servlet.view.JstlView  
hello.url = /WEB-INF/jsp/hello.jsp

For example, using the above configuration, if URI −

* /hello is requested, DispatcherServlet will forward the request to the hello.jsp defined by bean hello in the views.properties.
* Here, "hello" is the view name to be matched. Whereas, class refers to the view type and URL is the view's location.

**Configure multiple View Resolvers together**

In case you want to use a Multiple View Resolver in a Spring MVC application then priority order can be set using the order property. The following example shows how to use the ResourceBundleViewResolver, the InternalResourceViewResolver and the XmlViewResolver in the Spring Web MVC Framework.

## ***mvc-dispatcher-servlet.xml***

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation=" http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix">

<value>/WEB-INF/</value>

</property>

<property name="suffix">

<value>.jsp</value>

</property>

<property name="order" value="2" />

</bean>

<bean class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="location">

<value>/WEB-INF/views.xml</value>

</property>

<property name="order" value="1" />

</bean>

<bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename" value="views" />

<property name="order" value="0" />

</bean>

</beans>

Here, the order property defines the ranking of a view resolver. In this, 0 is the first resolver and 1 is the next resolver and so on.

## **views.properties**

hello.(class) = org.springframework.web.servlet.view.JstlView  
hello.url = /WEB-INF/jsp/hello.jsp

For example, using the above configuration, if URI −

* /hello is requested, DispatcherServlet will forward the request to the hello.jsp defined by bean hello in views.properties.

To start with, let us have a working Eclipse IDE in place and consider the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

**UrlBasedViewResolver**

The UrlBasedViewResolver is provides the mapping logical view names and URLs directly that hands over to the view class specified. The UrlBasedViewResolver provides a convenient shape called InternalResourceViewResolver that support JSP, Servlet, JstlViews and TileViews.

<bean id=*"viewResolver"* class=*"org.springframework.web.servlet.view.UrlBasedViewResolver"*>

<property name=*"viewClass"* value=*"org.springframework.web.servlet.view.JstlView"*/>

<property name=*"prefix"* value=*"/WEB-INF/jsp/"*/>

<property name=*"suffix"* value=*".jsp"*/>

</bean>

**Spring Annotation:**

**@RequestMapping ANNOTATION**

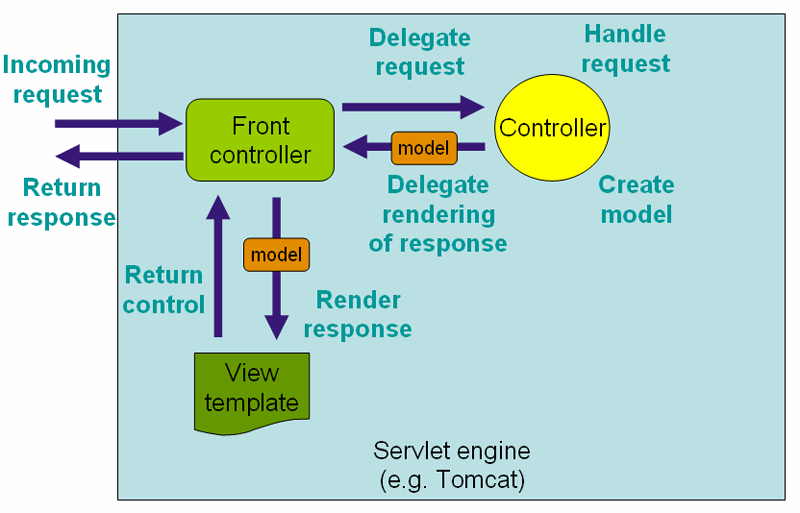
@RequestMapping is one of the most common annotation used in Spring Web applications. This annotation maps HTTP requests to handler methods of MVC and REST controllers.

In this post, you’ll see how versatile the @RequestMapping annotation is when used to map Spring MVC controller methods.

## **Request Mapping Basics**

In Spring MVC applications, the RequestDispatcher (Front Controller Below) servlet is responsible for routing incoming HTTP requests to handler methods of controllers.

When configuring Spring MVC, you need to specify the mappings between the requests and handler methods.



To configure the mapping of web requests, you use the @RequestMapping annotation.

The @RequestMapping annotation can be applied to class-level and/or method-level in a controller.

The class-level annotation maps a specific request path or pattern onto a controller. You can then apply additional method-level annotations to make mappings more specific to handler methods.

Here is an example of the @RequestMapping annotation applied to both class and methods.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping("/")

String get(){

//mapped to hostname:port/home/

return "Hello from get";

}

@RequestMapping("/index")

String index(){

//mapped to hostname:port/home/index/

return "Hello from index";

}

}

With the preceding code, requests to /home will be handled by get() while request to /home/index will be handled by index().

## @RequestMapping with Multiple URIs

You can have multiple request mappings for a method. For that add one @RequestMapping annotation with a list of values.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value={"", "/page", "page\*","view/\*,\*\*/msg"})

String indexMultipleMapping(){

return "Hello from index multiple mapping.";

}

}

As you can see in this code, @RequestMapping supports wildcards and ant-style paths. For the preceding code, all these URLs will be handled by indexMultipleMapping().

* localhost:8080/home
* localhost:8080/home/
* localhost:8080/home/page
* localhost:8080/home/pageabc
* localhost:8080/home/view/
* localhost:8080/home/view/view

## @RequestMapping with @RequestParam

The @RequestParam annotation is used with @RequestMapping to bind a web request parameter to the parameter of the handler method.

The @RequestParam annotation can be used with or without a value. The value specifies the request param that needs to be mapped to the handler method parameter, as shown in this code snippet.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/id")

String getIdByValue(@RequestParam("id") String personId){

System.out.println("ID is "+personId);

return "Get ID from query string of URL with value element";

}

@RequestMapping(value = "/personId")

String getId(@RequestParam String personId){

System.out.println("ID is "+personId);

return "Get ID from query string of URL without value element";

}

}

In Line 6 of this code, the request param id will be mapped to the personId parameter personId of thegetIdByValue() handler method.

An example URL is this:

localhost:8090/home/id?id=5

The value element of @RequestParam can be omitted if the request param and handler method parameter names are same, as shown in Line 11.

An example URL is this:

localhost:8090/home/personId?personId=5

The required element of @RequestParam defines whether the parameter value is required or not.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/name")

String getName(@RequestParam(value = "person", required = false) String personName){

return "Required element of request param";

}

}

In this code snippet, as the required element is specified as false, the getName() handler method will be called for both of these URLs:

* /home/name?person=xyz
* /home/name

The default value of the @RequestParam is used to provide a default value when the request param is not provided or is empty.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/name")

String getName(@RequestParam(value = "person", defaultValue = "John") String personName ){

return "Required element of request param";

}

}

In this code, if the person request param is empty in a request, the getName() handler method will receive the default value John as its parameter.

## Using @RequestMapping with HTTP Method

The Spring MVC @RequestMapping annotation is capable of handling HTTP request methods, such as GET, PUT, POST, DELETE, and PATCH.

By default all requests are assumed to be of HTTP GET type.

In order to define a request mapping with a specific HTTP method, you need to declare the HTTP method in@RequestMapping using the method element as follows.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(method = RequestMethod.GET)

String get(){

return "Hello from get";

}

@RequestMapping(method = RequestMethod.DELETE)

String delete(){

return "Hello from delete";

}

@RequestMapping(method = RequestMethod.POST)

String post(){

return "Hello from post";

}

@RequestMapping(method = RequestMethod.PUT)

String put(){

return "Hello from put";

}

@RequestMapping(method = RequestMethod.PATCH)

String patch(){

return "Hello from patch";

}

}

In the code snippet above, the method element of the @RequestMapping annotations indicates the HTTP method type of the HTTP request.

All the handler methods will handle requests coming to the same URL ( /home), but will depend on the HTTP method being used.

For example, a POST request to /home will be handled by the post() method. While a DELETE request to/home will be handled by the delete() method.

You can see how Spring MVC will map the other methods using this same logic.

## Using @RequestMapping with Producible and Consumable

The request mapping types can be narrowed down using the produces and consumes elements of the@RequestMapping annotation.

In order to produce the object in the requested media type, you use the produces element of@RequestMapping in combination with the @ResponseBody annotation.

You can also consume the object with the requested media type using the consumes element of@RequestMapping in combination with the @RequestBody annotation.

The code to use producible and consumable with @RequestMapping is this.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/prod", produces = {"application/JSON"})

@ResponseBody

String getProduces(){

return "Produces attribute";

}

@RequestMapping(value = "/cons", consumes = {"application/JSON", "application/XML"})

String getConsumes(){

return "Consumes attribute";

}

}

In this code, the getProduces() handler method produces a JSON response. The getConsumes() handler method consumes JSON as well as XML present in requests.

## @RequestMapping with Headers

The @RequestMapping annotation provides a header element to narrow down the request mapping based on headers present in the request.

You can specify the header element as myHeader = myValue.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/head", headers = {"content-type=text/plain"})

String post(){

return "Mapping applied along with headers";

}

}

In the above code snippet, the headers attribute of the @RequestMapping annotation narrows down the mapping to the post() method. With this, the post() method will handle requests to /home/head whosecontent-type header specifies plain text as the value.

You can also indicate multiple header values like this:

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/head", headers = {"content-type=text/plain", "content-type=text/html"}) String post(){

return "Mapping applied along with headers";

}

}

Here it implies that both text/plain as well as text/html are accepted by the post() handler method.

## @RequestMapping with Request Parameters

The params element of the @RequestMapping annotation further helps to narrow down request mapping. Using the params element, you can have multiple handler methods handling requests to the same URL, but with different parameters.

You can define params as myParams = myValue. You can also use the negation operator to specify that a particular parameter value is not supported in the request.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/fetch", params = {"personId=10"})

String getParams(@RequestParam("personId") String id){

return "Fetched parameter using params attribute = "+id;

}

@RequestMapping(value = "/fetch", params = {"personId=20"})

String getParamsDifferent(@RequestParam("personId") String id){

return "Fetched parameter using params attribute = "+id;

}

}

In this code snippet, both the getParams() and getParamsDifferent() methods will handle requests coming to the same URL ( /home/fetch) but will execute depending on the params element.

For example, when the URL is /home/fetch?id=10 the getParams() handler method will be executed with the id value 10.. For the URL, localhost:8080/home/fetch?personId=20, the getParamsDifferent() handler method gets executed with the id value 20.

## Using @RequestMapping with Dynamic URIs

The @RequestMapping annotation is used in combination with the @PathVaraible annotation to handle dynamic URIs. In this use case, the URI values can act as the parameter of the handler methods in the controller. You can also use regular expressions to only accept the dynamic URI values that match the regular expression

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/fetch/{id}", method = RequestMethod.GET)

String getDynamiIn this code, the method getDynamicUriValue() will execute for a request to localhost:8080/home/fetch/10. Also, the id parameter of the getDynamicUriValue() handler method will be populated with the value 10dynamically.

The method getDynamicUriValueRegex() will execute for a request to localhost:8080/home/fetch/category/shirt. However, an exception will be thrown for a request to /home/fetch/10/shirt as it does not match the regular expression.

@PathVariable works differently from @RequestParam. You use @RequestParam to obtain the values of the query parameters from the URI. On the other hand, you use @PathVariable to obtain the parameter values from the URI template.

## The @RequestMapping Default Handler Method

In the controller class you can have default handler method that gets executed when there is a request for a default URI.

Here is an example of a default handler method.

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping()

String default(){

return "This is a default method for the class";

}

}

In this code, A request to /home will be handled by the default() method as the annotation does not specify any value.

## @RequestMapping Shortcuts

Spring 4.3 introduced method-level variants, also known as composed annotations of @RequestMapping. The composed annotations better express the semantics of the annotated methods. They act as wrapper to@RequestMapping and have become the standard ways of defining the endpoints.

For example, @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method =RequestMethod.GET).

The method level variants are:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

The following code shows using the composed annotations.

@RestController

@RequestMapping("/home")

public class IndexController {

@GetMapping("/person")

public @ResponseBody ResponseEntity<String> getPerson() {

return new ResponseEntity<String>("Response from GET", HttpStatus.OK);

}

@GetMapping("/person/{id}")

public @ResponseBody ResponseEntity<String> getPersonById(@PathVariable String id){

return new ResponseEntity<String>("Response from GET with id " +id,HttpStatus.OK); }

@PostMapping("/person")

public @ResponseBody ResponseEntity<String> postPerson() {

return new ResponseEntity<String>("Response from POST method", HttpStatus.OK);

}

@PutMapping("/person")

public @ResponseBody ResponseEntity<String> putPerson() {

return new ResponseEntity<String>("Response from PUT method", HttpStatus.OK);

}

@DeleteMapping("/person")

public @ResponseBody ResponseEntity<String> deletePerson() {

return new ResponseEntity<String>("Response from DELETE method", HttpStatus.OK);

}

@PatchMapping("/person")

public @ResponseBody ResponseEntity<String> patchPerson() {

return new ResponseEntity<String>("Response from PATCH method", HttpStatus.OK);

}

}

In this code, each of the handler methods are annotated with the composed variants of @RequestMapping. Although, each variant can be interchangeably used with @RequestMapping with the method attribute, it’s considered a best practice to use the composed variant. Primarily because the composed annotations reduce the configuration metadata on the application side and the code is more readable.

## @RequestMapping Conclusion

As you can see in this post, the @RequestMapping annotation is very versatile. You can use this annotation to configure Spring MVC to handle a variety of use cases. It can be used to configure traditional web page requests, and well as RESTFul web services in Spring MVC.