# 第五部分 Web

This part of the reference documentation covers Spring Framework’s support for the presentation tier (and specifically web-based presentation tiers) including support for WebSocket-style messaging in web applications.

Spring Framework’s own web framework, [Spring Web MVC](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc), is covered in the first couple of chapters. Subsequent chapters are concerned with Spring Framework’s integration with other web technologies, such as [JSF](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#jsf).

The section then concludes with comprehensive coverage of the Spring Framework [Chapter 22, *WebSocket Support*](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#websocket) (including [Section 22.4, “STOMP Over WebSocket Messaging Architecture”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#websocket-stomp)).

* [Chapter 18, *Web MVC 框架*](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc)
* Chapter 19, *视图技术*
* [Chapter 21, *集成其他的Web框架*](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#web-integration)
* [Chapter 22, *WebSocket支持*](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#websocket)

## 18. Web MVC框架

### 18.1 Spring Web MVC 框架介绍

Spring Web MVC框架是围绕一个用于把请求分发到处理方法的DispatcherServlet来设计的，它还包括可配置的处理方法映射、视图解析、本地化、时区和主题解析以及文件上传支持。默认的处理程序是局域注解@Controller和@RequestMapping的，提供一个大范围的、灵活的处理方法。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.

"Open for extension…​" A key design principle in Spring Web MVC and in Spring in general is the "*Open for extension, closed for modification*" principle.

Some methods in the core classes of Spring Web MVC are marked final. As a developer you cannot override these methods to supply your own behavior. This has not been done arbitrarily, but specifically with this principle in mind.

For an explanation of this principle, refer to *Expert Spring Web MVC and Web Flow* by Seth Ladd and others; specifically see the section "A Look At Design," on page 117 of the first edition. Alternatively, see

* [Bob Martin, The Open-Closed Principle (PDF)](https://www.cs.duke.edu/courses/fall07/cps108/papers/ocp.pdf)

You cannot add advice to final methods when you use Spring MVC. For example, you cannot add advice to the AbstractController.setSynchronizeOnSession() method. Refer to [Section 7.6.1, “Understanding AOP proxies”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#aop-understanding-aop-proxies) for more information on AOP proxies and why you cannot add advice to final methods.

In Spring Web MVC you can use any object as a command or form-backing object; you do not need to implement a framework-specific interface or base class. Spring’s data binding is highly flexible: for example, it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. Thus you do not need to duplicate your business objects' properties as simple, untyped strings in your form objects simply to handle invalid submissions, or to convert the Strings properly. Instead, it is often preferable to bind directly to your business objects.

Spring’s view resolution is extremely flexible. 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. View name resolution is highly configurable through file extension or Accept header content type negotiation, through bean names, a properties file, or even a custom ViewResolver implementation. The model (the M in MVC) is a Map interface, which allows for the complete abstraction of the view technology. You can integrate directly with template based rendering technologies such as JSP and FreeMarker, or directly generate XML, JSON, Atom, and many other types of content. The model Map is simply transformed into an appropriate format, such as JSP request attributes or a FreeMarker template model.

#### 18.1.1 Spring Web MVC 的特性

**Spring Web流**

Spring Web流（Spring Web Flow：SWF）的目标是成为最好的解决方案来管理Web应用程序的页面流。

SWF在Servlet和Portlet环境中和已经存在的框架比如Spring MVC给和JSF集成。If you have a business process (or processes) that would benefit from a conversational model as opposed to a purely request model, then SWF may be the solution.

SWF allows you to capture logical page flows as self-contained modules that are reusable in different situations, and as such is ideal for building web application modules that guide the user through controlled navigations that drive business processes.

For more information about SWF, consult the [Spring Web Flow website](http://projects.spring.io/spring-webflow/).

Spring’s web module includes many unique web support features:

* *明确的角色分工（Clear separation of roles）*. Each role — controller, validator, command object, form object, model object, DispatcherServlet, handler mapping, view resolver, and so on — can be fulfilled by a specialized object.
* *Powerful and straightforward configuration of both framework and application classes as JavaBeans*. This configuration capability includes easy referencing across contexts, such as from web controllers to business objects and validators.
* *Adaptability, non-intrusiveness, and flexibility.* Define any controller method signature you need, possibly using one of the parameter annotations (such as @RequestParam, @RequestHeader, @PathVariable, and more) for a given scenario.
* *Reusable business code, no need for duplication*. Use existing business objects as command or form objects instead of mirroring them to extend a particular framework base class.
* *Customizable binding and validation*. Type mismatches as application-level validation errors that keep the offending value, localized date and number binding, and so on instead of String-only form objects with manual parsing and conversion to business objects.
* *Customizable handler mapping and view resolution*. Handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. Spring is more flexible than web MVC frameworks that mandate a particular technique.
* *Flexible model transfer*. Model transfer with a name/value Map supports easy integration with any view technology.
* *Customizable locale, time zone and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for FreeMarker without the need for extra bridges, and so on.*
* *A simple yet powerful JSP tag library known as the Spring tag library that provides support for features such as data binding and themes*. The custom tags allow for maximum flexibility in terms of markup code. For information on the tag library descriptor, see the appendix entitled [Chapter 40, *spring JSP Tag Library*](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#spring-tld)
* *A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier.* For information on the tag library descriptor, see the appendix entitled [Chapter 41, *spring-form JSP Tag Library*](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#spring-form-tld)
* *Beans whose lifecycle is scoped to the current HTTP request or HTTP Session.* This is not a specific feature of Spring MVC itself, but rather of the WebApplicationContext container(s) that Spring MVC uses. These bean scopes are described in [Section 3.5.4, “Request, session, application, and WebSocket scopes”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#beans-factory-scopes-other)

#### 18.1.2 Pluggability of other MVC implementations

Non-Spring MVC implementations are preferable for some projects. Many teams expect to leverage their existing investment in skills and tools, for example with JSF.

If you do not want to use Spring’s Web MVC, but intend to leverage other solutions that Spring offers, you can integrate the web MVC framework of your choice with Spring easily. Simply start up a Spring root application context through its ContextLoaderListener, and access it through its ServletContext attribute (or Spring’s respective helper method) from within any action object. No "plug-ins" are involved, so no dedicated integration is necessary. From the web layer’s point of view, you simply use Spring as a library, with the root application context instance as the entry point.

Your registered beans and Spring’s services can be at your fingertips even without Spring’s Web MVC. Spring does not compete with other web frameworks in this scenario. It simply addresses the many areas that the pure web MVC frameworks do not, from bean configuration to data access and transaction handling. So you can enrich your application with a Spring middle tier and/or data access tier, even if you just want to use, for example, the transaction abstraction with JDBC or Hibernate.

### 18.2 DispatcherServlet

Spring MVC框架像很多其他的MVC框架那样，是请求驱动的、围绕一个中心Servlet设计的，这个Servlet分发请求到到控制器并提供其他有助于开发Web应用程序的功能。不过Spring的DispatcherServlet能做的可是不止这些。它完全与 Spring的IoC容器集成，并且允许你使用Spring所拥有的所有其他特性。

Spring Web MVC DispatcherServlet的请求处理工作流的说明见下图。The pattern-savvy reader will recognize that the DispatcherServlet is an expression of the "Front Controller" design pattern (this is a pattern that Spring Web MVC shares with many other leading web frameworks).

**Figure 18.1. The request processing workflow in Spring Web MVC (high level)**



DispatcherServlet事实上是一个Servlet（它继承自基类HttpServlet），所以它可以在你的Web应用中声明。你需要通过一个URL mapping来映射你想让DispatcherServlet处理的请求。这是一个在Servlet 3.0+环境下的标准的Java EE Servlet配置：

public class MyWebApplicationInitializer implements WebApplicationInitializer

{

*@Override*

public void onStartup(ServletContext container)

{

ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet());

registration.setLoadOnStartup(1);

registration.addMapping("/example/\*");

}

}

在上面的例子中，所有以“/example”开头的请求都会被DispatcherServlet实例所处理。

WebApplicationInitializer是一个由Spring MVC提供的接口，用于保证你的基于代码的配置被检测到并自动初始化到任何Servlet 3容器。 这个接口的一个抽象基类实现是AbstractAnnotationConfigDispatcherServletInitializer使得注册DispatcherServlet更加容易，只需简单地指定它的servlet mapping并列出配置类即可，这是构建你的Spring MVC应用程序的推荐方式。详情参见[基于代码的Servlet容器初始化](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-container-config)。

DispatcherServlet事实上是一个Servlet（它继承自基类HttpServlet），所以它可以在你的Web应用中的web.xml中声明。你需要通过一个在同一个web.xml文件中的URL mapping来映射你想让DispatcherServlet处理的请求。这是标准的Java EE Servlet配置；下面的例子展示了这样一个DispatcherServlet声明与配置：

下面的web.xml等价于上面的基于代码的配置示例：

<web-app>

<servlet>

<servlet-name>example</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>example</servlet-name>

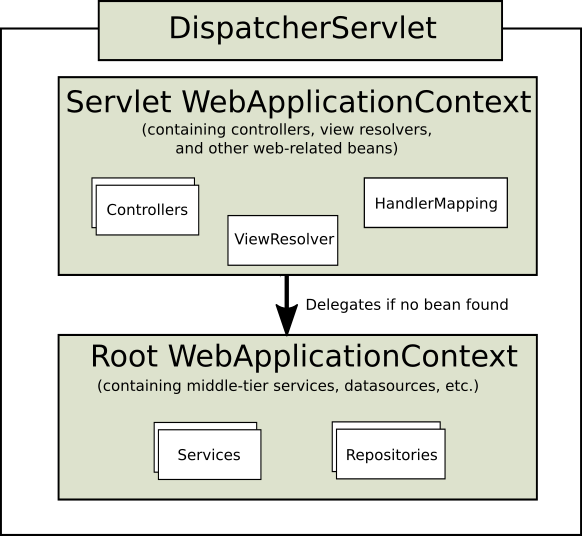
<url-pattern>/example/\*</url-pattern>

</servlet-mapping>

</web-app>

就像[3.15节 ApplicationContext的额外功能](#_3.15 ApplicationContext的额外功能)中详细描述的一样，Spring中ApplicationContext实例是可以限定作用域的。在Web MVC框架中，每一个DispatcherServlet都有自己的WebApplicationContext，它继承已经定义在根WebApplicationContext中的所有的Bean。这个根WebApplicationContext应该包含所有的要分享到其他上下文和Servlet实例中的基本Bean。这些被继承的Bean可以在指定的Servlet作用域中重写，你也可以为一个给定的Servlet实例定义新的指定作用域的本地Bean来。

**Figure 18.2. Typical context hierarchy in Spring Web MVC**



在初始化DispatcherServlet的时候，Spring MVC在你的应用程序的WEB-INF目录下查找一个叫做*[servlet-name]-servlet.xml*的文件，并创建里面定义的Bean，覆盖任何定义在全局作用域中的同名的Bean。

考虑下面的DispatcherServlet Servlet配置（在web.xml文件中）：

<web-app>

<servlet>

<servlet-name>**golfing**</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>**golfing**</servlet-name>

<url-pattern>/golfing/\*</url-pattern>

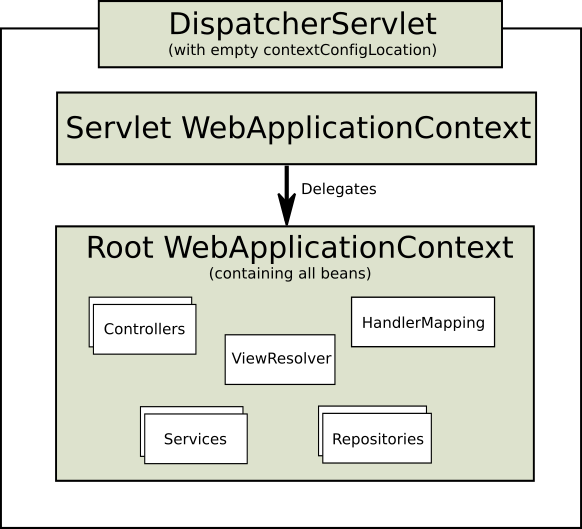
</servlet-mapping>

</web-app>

对于上面的Servlet配置，你的应用中需要一个名叫***/WEB-INF/golfing-servlet.xml***的文件；这个文件会包含你的Spring Web MVC指定的所有组件（Bean）。你可以通过一个Servlet初始化擦书来改变这个配置文件的位置（详情见下面）。

It is also possible to have just one root context for single DispatcherServlet scenarios.

**Figure 18.3. Single root context in Spring Web MVC**



This can be configured by setting an empty contextConfigLocation servlet init parameter, as shown below:

<web-app>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/root-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

</web-app>

WebApplicationContext是一个ApplicationContext的扩展，它包含一些Web应用程序必须的额外特性。It differs from a normal ApplicationContext in that it is capable of resolving themes (see [Section 18.9, “Using themes”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-themeresolver)), and that it knows which Servlet it is associated with (by having a link to the ServletContext). The WebApplicationContext is bound in the ServletContext, and by using static methods on the RequestContextUtils class you can always look up the WebApplicationContext if you need access to it.

注意，我们可以用基于Java的配置实现同样的功能：

public class GolfingWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer

{

*@Override*

protected Class<?>[] getRootConfigClasses()

{

return new Class[] { GolfingAppConfig.class }; // GolfingAppConfig defines beans that would be in root-context.xml

}

*@Override*

protected Class<?>[] getServletConfigClasses()

{

return new Class[] { GolfingWebConfig.class }; // GolfingWebConfig defines beans that would be in golfing-servlet.xml

}

*@Override*

protected String[] getServletMappings() { return new String[] { "/golfing/\*" }; }

}

#### 18.2.1 WebApplicationContext中的特殊Bean类型

Spring DispatcherServlet使用特殊的Bean来处理请求并渲染适当的视图。这些Bean是Spring MVC的一部分。你可以通过在WebApplicationContext中简单地配置一个或多个Bean来选择使用哪些特殊的Bean。然而，你不需要自己来做，因为就算你不配置任何的特殊Bean，Spring MVC也维护了一个默认的Bean列表来用。More on that in the next section. First see the table below listing the special bean types the DispatcherServlet relies on.

**表 18.1. WebApplicationContext中的特殊Bean类型**

| **Bean 类型** | **解释** |
| --- | --- |
| [HandlerMapping](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-handlermapping) | Maps incoming requests to handlers and a list of pre- and post-processors (handler interceptors) based on some criteria the details of which vary by HandlerMapping implementation. The most popular implementation supports annotated controllers but other implementations exists as well. |
| HandlerAdapter | Helps the DispatcherServlet to invoke a handler mapped to a request regardless of the handler is actually invoked. For example, invoking an annotated controller requires resolving various annotations. Thus the main purpose of a HandlerAdapter is to shield the DispatcherServlet from such details. |
| [HandlerExceptionResolver](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-exceptionhandlers) | Maps exceptions to views also allowing for more complex exception handling code. |
| [ViewResolver](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-viewresolver) | Resolves logical String-based view names to actual View types. |
| [LocaleResolver](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-localeresolver)&[LocaleContextResolver](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-timezone) | Resolves the locale a client is using and possibly their time zone, in order to be able to offer internationalized views |
| [ThemeResolver](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-themeresolver) | Resolves themes your web application can use, for example, to offer personalized layouts |
| [MultipartResolver](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-multipart) | Parses multi-part requests for example to support processing file uploads from HTML forms. |
| [FlashMapManager](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-flash-attributes) | Stores and retrieves the "input" and the "output" FlashMap that can be used to pass attributes from one request to another, usually across a redirect. |

#### 18.2.2 默认的DispatcherServlet配置

就像上一节提及的那样，对于每一个特殊Bean，DispatcherServlet维护了一个默认的实现列表来用。这个信息保存在包org.springframework.web.servlet下的DispatcherServlet.properties文件中。

所有的特殊Bean自己都有一些合理的默认值，尽管你迟早都会需要自定义一个或多个这些Bean提供的属性。比如，常见的是配置一个InternalResourceViewResolver设置它的prefix属性来定位视图文件的父路径。

抛开细节，这里需要理解的重要概念是，一旦你在WebApplicationContext应用中配置了一个特殊的Bean，比如一个InternalResourceViewResolver，你事实上是重写了要使用的、存放特殊类型Bean的默认实现的列表。比如，你配置了一个InternalResourceViewResolver，那么默认的ViewResolver实现列表就会被忽略。

In [Section 18.16, “Configuring Spring MVC”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config) you’ll learn about other options for configuring Spring MVC including MVC Java config and the MVC XML namespace both of which provide a simple starting point and assume little knowledge of how Spring MVC works. Regardless of how you choose to configure your application, the concepts explained in this section are fundamental and should be of help to you.

#### 18.2.3 DispatcherServlet处理序列

在你创建了一个DispatcherServlet之后，一个请求进入到这个指定的DispatcherServlet时，DispatcherServlet就开始像下面那样处理请求：

* 请求搜索它绑定的WebApplicationContext，这个请求用作控制器和其他元素处理过程中使用的一个属性。这个WebApplicationContext默认绑定在键DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE下面。
* 在处理请求（渲染视图、准备数据、等）时，绑定到请求的地区解析器使处理过程中的元素能够解析要使用的地区。如果你不需要解析地区，那你就不需要地区解析器。
* 绑定到请求的主题解析器让元素（比如视图）来决定使用哪一个主题。如果你不需要使用主题，那你就不需要主题解析器。
* 如果你指定一个多部分文件解析器（multipart file resolver），那么请求就检查multiparts；如果找到了multiparts，那么请求就被包装进MultipartHttpServletRequest用于其他元素以后的处理。详情参见 [18.10节- Spring的多个文件上传支持](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-multipart)。
* 查找一个适当的处理器。如果发现一个处理器，那么与处理器（预处理器preprocessors、后处理器postprocessors和控制器controllers）相关联的执行链会被执行，来准备一个模型和渲染。
* 如果模型被返回，就渲染视图。如果没有模型返回（可能因为预处理器或者后处理器拦截了请求，可能因为安全原因），就没有视图要渲染，因为请求已经被满足了。

声明在WebApplicationContext中的处理异常解析器拾取请求处理过程中抛出的异常。使用这些异常解析器，你可以定义自定义行为来处理异常。

Spring的DispatcherServlet也支持返回*最后修改时间*（*last-modification-date*）就像Servlet API指定的那样。为一个指定的请求决定最后修改时间是很直接的：DispatcherServlet查找一个合适的处理程序映射（handler mapping）并测试这个被发现的处理程序是否实现了*LastModified*接口。如果实现了，那么接口LastModified 中方法getLastModified(request)的返回值会被返回到客户端。

你可以通过在web.xml文件中添加Servlet初始化参数（init-param元素）来定制自己的DispatcherServlet实例。下表列出了支持的参数：

**表18.2. DispatcherServlet初始化参数**

| **参数** | **解释** |
| --- | --- |
| contextClass | Class that implements WebApplicationContext, which instantiates the context used by this Servlet. By default, the XmlWebApplicationContext is used. |
| contextConfigLocation | String that is passed to the context instance (specified by contextClass) to indicate where context(s) can be found. The string consists potentially of multiple strings (using a comma as a delimiter) to support multiple contexts. In case of multiple context locations with beans that are defined twice, the latest location takes precedence. |
| namespace | Namespace of the WebApplicationContext. Defaults to [servlet-name]-servlet. |

### 18.3 实现控制器

控制器提供访问你通过一个服务接口定义的应用程序行为。控制器解释用户输入，并把它转换到一个通过视图来表示用户的模型（model）中。Spring用一种非常抽象的方式实现一个控制器，这使得你可以创建多样的控制器。

Spring 2.5介绍了一种用于MVC控制器的基于注解的编程模型，它使用注解，比如@RequestMapping、@RequestParam、@ModelAttribute等。这个注解同时支持Servlet MVC和Portlet MVC。用这种方式实现的控制器，不需要集成任何指定的基类或者实现任何指定的接口。而且，它们通常不直接依赖于Servlet或Portlet的API，尽管你可以轻松的配置要访问的Servlet或Portlet。

|  |
| --- |
| [Tip] |
| Available in the [spring-projects Org on Github](https://github.com/spring-projects/), a number of web applications leverage the annotation support described in this section including *MvcShowcase*, *MvcAjax*, *MvcBasic*, *PetClinic*, *PetCare*, and others. |

*@Controller*

public class HelloWorldController

{

*@RequestMapping("/helloWorld")*

public String helloWorld(Model model)

{

model.addAttribute("message", "Hello World!");

return "helloWorld";

}

}

就像你看到的，注解@Controller和@RequestMapping允许灵活的方法名称和签名。在这个特定的例子中，方法接受一个Model并以字符串的形式返回一个视图名，但是，其他方法参数和返回值也是可用的，本节稍后会解释。@Controller、@RequestMapping和一些其他注解是建立在Spring MVC实现的基础上的。本节解释这些注解以及他们通常怎样在Servlet环境中使用。

#### 18.3.1 用@Controller定义一个控制器

注解@Controller指定一个特定的类担任*控制器*的角色。Spring不要求你集成任何控制器基类或者引用Servlet API。不过，你仍然可以根据需要使用指定的Servlet特性。

注解@Controller对于被注解的类来说就像一个模板（stereotype），指示它的角色。收发器（dispatcher）为被映射的方法扫描被注解的类，并检测注解@RequestMapping（见下一部分）。

你可以在分发器的上下文中使用标准的Spring Bean定义，来显式地定义被注解的控制器。不过，@Controller模板也允许自动检测，就像Spring通常支持的在类路径中自动检测组件类并自动为它们注册Bean定义。

为了能够自动检测到这样被注解的控制器，你要添加组件扫描到你的配置中。像下面的XML片段那样使用Spring的context'命名空间：

<?xml version="1.0" encoding="UTF-8"?>

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

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

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

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

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

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

<context:component-scan base-package="org.springframework.samples.petclinic.web"/>

<!-- ... -->

</beans>

#### 18.3.2 使用@RequestMapping映射请求

使用注解@RequestMapping映射一个URL（比如：/appointments）到一个类或者一个特定的处理方法上。典型地，类级别的注解映射一个指定的请求路径（或者是路径匹配模式）到一个形式控制器（a form controller），使用额外的方法层注解缩小主要映射的范围。

下面来自样例*Petcare*的例子展示了一个在Spring MVC应用程序中使用这个注解的控制器：

*@Controller*

**@RequestMapping("/appointments")**

public class AppointmentsController

{

private final AppointmentBook appointmentBook;

*@Autowired*

public AppointmentsController(AppointmentBook appointmentBook) { this.appointmentBook = appointmentBook; }

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

public Map<String, Appointment> get() { return appointmentBook.getAppointmentsForToday(); }

**@RequestMapping(path = "/{day}", method = RequestMethod.GET)**

public Map<String, Appointment> getForDay(*@PathVariable@DateTimeFormat(iso=ISO.DATE)* Date day, Model model)

{

return appointmentBook.getAppointmentsForDay(day);

}

**@RequestMapping(path = "/new", method = RequestMethod.GET)**

public AppointmentForm getNewForm() { return new AppointmentForm(); }

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

public String add(*@Valid* AppointmentForm appointment, BindingResult result)

{

if (result.hasErrors()) return "appointments/new";

appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

}

}

在上面的例子中，注解@RequestMapping被用在了多处。第一处是类级别的，它指出这个控制器中的所有方法都与路径/appointments相关。方法get()上有一个@RequestMapping来进一步细化：它只接受GET请求，这意味着HTTP GET请求/appointments将调用这个方法。方法add()有一个相似的细化；方法getNewForm()把HTTP方法定义和路径合并到了一起，这样GET请求/appointments/new就可以被这个方法处理了。

方法getForDay()展示了另一种使用@RequestMapping的方式：URI 模板。（见[“URI模板模式](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-requestmapping-uri-templates)”一节）

类级别的@RequestMapping不是必须的。没有它，所有的路径都是简单的绝对路径，而不是相对的。下面来自例程*PetClinic*的例子展示了一个使用@RequestMapping的多动作（multi-action）控制器：

*@Controller*

public class ClinicController

{

private final Clinic clinic;

*@Autowired*

public ClinicController(Clinic clinic) { this.clinic = clinic; }

**@RequestMapping("/")**

public void welcomeHandler() { }

**@RequestMapping("/vets")**

public ModelMap vetsHandler() { eturn new ModelMap(this.clinic.getVets()); }

}

上面的例子没有指定GET、PUT、POST等，@RequestMapping默认映射所有的HTTP方法。使用@RequestMapping(method=GET)或者@GetMapping来细化映射。

##### 组合的@RequestMapping变种

Spring 4.3中引进了下面的注解@RequestMapping在方法层级的变种，来帮助简化常用HTTP方法的映射，并更好地表达被注解的方法的语义。比如，@GetMapping可以读作GET @RequestMapping。

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

下面的例子展示了上一节中AppointmentsController 的一个修改版本，它使用了简化过的组合的@RequestMapping注解：

*@Controller*

**@RequestMapping("/appointments")**

public class AppointmentsController

{

private final AppointmentBook appointmentBook;

*@Autowired*

public AppointmentsController(AppointmentBook appointmentBook) { this.appointmentBook = appointmentBook; }

**@GetMapping**

public Map<String, Appointment> get() { return appointmentBook.getAppointmentsForToday(); }

**@GetMapping("/{day}")**

public Map<String, Appointment> getForDay(*@PathVariable@DateTimeFormat(iso=ISO.DATE)* Date day, Model model)

{

return appointmentBook.getAppointmentsForDay(day);

}

**@GetMapping("/new")**

public AppointmentForm getNewForm() { return new AppointmentForm(); }

**@PostMapping**

public String add(*@Valid* AppointmentForm appointment, BindingResult result)

{

if (result.hasErrors()) return "appointments/new";

appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

}

}

##### @Controller和AOP代理

在一些情形下，控制器在运行时可能需要AOP代理来修饰。比如当你选择直接在控制器上添加@Transactional注解时。当出现这样的情况时，对于控制器，我们建议使用基于类的代理，这是默认的选择。如果一个控制器必须实现一个不是Spring上下文回调的接口，你可能需要显式地配置基于类的代理。比如，对于<tx:annotation-driven/>，应该改为<tx:annotation-driven proxy-target-class="true"/>。

##### Spring MVC 3.1中@RequestMapping方法的新的支持类

Spring 3.1引进了一组新的@RequestMapping方法支持类，它们是RequestMappingHandlerMapping和RequestMappingHandlerAdapter。它们推荐使用，当然也要求使用Spring MVC 3.1以及后续版本中的新特性。这些新的支持类默认可以在MVC命名空间或者MVC Java配置中使用，但是如果都不适用的话，就要显式地进行配置。这一节描述新的和老的支持类之间的一个重要的不同之处。

在Spring 3.1之前，类型和方法级别的请求映射在两个不同的阶段进行检测——首先在控制器被DefaultAnnotationHandlerMapping选择的时候，然后在方法被AnnotationMethodHandlerAdapter收窄找到并调用的时候。

对于Spring 3.1中的新的支持类来说，RequestMappingHandlerMapping是唯一一处决定哪个方法用来处理请求的地方。请认为控制器方法是一个从类型和方法级别的@RequestMapping信息中映射来的不重复的端点集合。

这提供了一些新的可能性。这一次 HandlerInterceptor 或 HandlerExceptionResolver 现在可以预期基于对象的处理程序是 HandlerMethod，这允许他们精确第检查方法、方法参数以及方法上的注解。对一个URL的处理不再需要被分到不同的控制器中。

There are also several things no longer possible:

* Select a controller first with a SimpleUrlHandlerMapping or BeanNameUrlHandlerMapping and then narrow the method based on @RequestMapping annotations.
* Rely on method names as a fall-back mechanism to disambiguate between two @RequestMapping methods that don’t have an explicit path mapping URL path but otherwise match equally, e.g. by HTTP method. In the new support classes @RequestMapping methods have to be mapped uniquely.
* Have a single default method (without an explicit path mapping) with which requests are processed if no other controller method matches more concretely. In the new support classes if a matching method is not found a 404 error is raised.

The above features are still supported with the existing support classes. However to take advantage of new Spring MVC 3.1 features you’ll need to use the new support classes.

##### URI模板模式

*URI模板* 可以很方便地用于访问一个@RequestMapping方法中的URL的被选择的部分。

URI模板是一个像URI的字符串，它包含一个或多个变量名。当你用值来替代这些变量时，这个模板就会变成一个URI。为URI模板拟定的[RFC](http://bitworking.org/projects/URI-Templates/) 文档定义了一个URI是怎样被参数化的。比如，URI模板[http://www.example.com/users/{userId}](http://www.example.com/users/%7BuserId%7D)包含一个变量*userId*。把值*fred*赋到这个变量中就产生了<http://www.example.com/users/fred>。

在Spring MVC中，你可是在一个方法参数上使用注解@PathVariable，来把这个参数绑定到一个URI模板变量上：

*@GetMapping("/owners/{ownerId}")*

public String findOwner(**@PathVariable** String ownerId, Model model)

{

Owner owner = ownerService.findOwner(ownerId);

model.addAttribute("owner", owner);

return "displayOwner";

}

URI模板“/owners/{ownerId}”指定了一个名叫ownerId的变量。当控制器处理这个请求时，ownerId的值被设置为从URI中解析出来的合适的部分。比如，当请求 /owners/fred 进来时，fred就是ownerId的值。

|  |
| --- |
| 为了处理注解 @PathVariable，Spring MVC需要按名找到匹配URI模板的变量。你可以在这个注解中指定：  *@GetMapping("/owners/{ownerId}")*  public String findOwner(**@PathVariable("ownerId")** String theOwner, Model model)  {  // implementation omitted  }  如果URI模板变量名匹配方法参数名，那么你可以省略详细配置。只要你的代码在编译时带有调试信息或者在使用Java 8编译时使用参数 -parameters，Spring MVC就会匹配方法参数名到URI模板变量名：  *@GetMapping("/owners/{ownerId}")*  public String findOwner(**@PathVariable** String ownerId, Model model)  {  // implementation omitted  } |

一个方法可以有任何数量的注解@PathVariable：

*@GetMapping("/owners/{ownerId}/pets/{petId}")*

public String findPet(**@PathVariable** String ownerId, **@PathVariable** String petId, Model model)

{

Owner owner = ownerService.findOwner(ownerId);

Pet pet = owner.getPet(petId);

model.addAttribute("pet", pet);

return "displayPet";

}

当在一个Map<String, String>参数上使用@PathVariable注解时，这个Map会被填上所有的URI模板变量。

一个URI模板可以被集成到类和方法级别的*@RequestMapping*注解上。方法findPet()可以被一个 /owners/42/pets/21这样的URI调用。

*@Controller*

@RequestMapping(**"/owners/{ownerId}"**)

public class RelativePathUriTemplateController

{

@RequestMapping(**"/pets/{petId}"**)

public void findPet(*@PathVariable* String ownerId, *@PathVariable* String petId, Model model)

{

// implementation omitted

}

}

一个@PathVariable参数可以是*任何简单类型*，比如int，long，Date等。Spring自动转换到合适的类型，如果转换失败，就抛出一个TypeMismatchException异常。你也可以注册自己的数据类型转换支持。参见[“方法参数和类型转换”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-typeconversion)和[“自定义WebDataBinder初始化”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-webdatabinder)。

##### URI模板模式和正则表达式

有时，在定义URI模板变量时，你需要更高的精确度。考虑URL“/spring-web/spring-web-3.0.5.jar”。你怎么把它分成几部分呢？

注解@RequestMapping支持在URI模板变量中使用正则表达式。语法{varName:regex}的第一部分定义了变量名，第二部分是一个正则表达式。例如：

*@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]+}")*

public void handle(*@PathVariable* String version, *@PathVariable* String extension)

{

// ...

}

##### 路径模式

除了URI模板，注解@RequestMapping和所有的组合@RequestMapping变种也支持Ant式的路径模式（例如： /myPath/\*.do）。URI模板变量和Ant式的标记组合使用也是支持的（例如：/owners/\*/pets/{petId}）。

##### 路径模式比较

当一个URL匹配多个模式时，会进行一次分类来寻找最佳匹配。

***拥有数量最少的URI变量和通配符的匹配的路径模式被认为是最佳匹配***。比如“/hotels/{hotel}/\*”有一个URI变量和一个通配符，而“/hotels/{hotel}/\*\*”有一个URI变量和量个通配符，所以前者被认为是最佳匹配。

***如果两个模式有相同数量的通配符或URI变量，那么最长的那个被认为是最佳匹配***。比如“/foo/bar\*”比“/foo/\*”长，所以前者是最佳匹配。

***如果两个模式有相同数量的通配符或URI变量，且长度相同，那么通配符最少的那个被认为是最佳匹配***。比如“/hotels/{hotel}”相对于“/hotels/\*”是最佳匹配。

还有一些额外的特殊规则：

* **默认的映射模式**“*/*\*\*”比任何其他匹配模式的优先级都低。比如：“/api/{a}/{b}/{c}”的优先级高于“*/*\*\*”
* **前缀模式**比如“/public/\*\*”的优先级低于任何其他的不包含双通配符的模式。比如“/public/path3/{a}/{b}/{c}”的优先级更高。

全部的详细信息参见AntPathMatcher中的AntPatternComparator。注意，PathMatcher可以自定义（见[18.16.11节 “路径匹配”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-path-matching)）。

##### 带有占位符的路径模式

注解@RequestMapping中的模式支持对本地属性和/或系统属性与环境变量使用${…​}占位符。这在控制器映射到的路径需要通过配置文件自定义的情形下会很有用。占位符的更多信息，可以去类PropertyPlaceholderConfigurer的Javadoc中查看。

##### 后缀模式匹配

Spring MVC默认执行“.\*”后缀匹配，所以一个映射到“/person”的控制器隐式地映射到“/person.\*”。这让请求通过URL路径（比如person.pdf，/person.xml）来获取资源很简单。

后缀模式匹配可以被关闭，或者把它限制到一组显式注册为协商好内容的路径扩展。这通常建议用于在普通请求映射中减少歧义，比如“/person/{id}”中的一个点可能不代表文件扩展名，如“/person/joe@email.com”和“/person/joe@email.com.json”。而且，就像下面解释的那样，后缀模式匹配和内容协商可能被用在一些情形中来企图进行恶意攻击，所以有很好的理由来限制它们。

后缀匹配配置参见[18.16.11节“路径匹配”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-path-matching) ，内容协商配置参见 [18.16.6节“内容协商”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-content-negotiation)。

##### 后缀模式匹配和RFD

反射型文件下载（Reflected file download：RFD）攻击在2014年最初由Trustwave在一篇[论文](https://www.trustwave.com/Resources/SpiderLabs-Blog/Reflected-File-Download---A-New-Web-Attack-Vector/)中描述。这种袭击是类似于 XSS，因为它依赖于反映在响应中的输入（如查询参数，URI 变量）。不像在HTML中插入JavaScript，一次RFD攻击依赖于浏览器执行一个下载并把响应当作可执行脚本来处理，就像双击了像拥有扩展名.bat、.cmd的文件。

在Spring MVC中@ResponseBody和ResponseEntity的方法面临危险，因为它们可以渲染客户端能够通过URL路径扩展请求的不同类型的内容。注意到即不禁止后缀模式匹配也不单纯为内容协商目的禁止使用文件拓展名将有效防止RFD攻击。

为了全面防范RFD攻击，在渲染响应体之前，Spring MVC添加了一个Content-Disposition:inline;filename=f.txt响应头来暗示一个修正过的安全的下载文件。这仅在URL路径包含一个文件拓展名既不在白名单中，又没有因内容协商目的而显式注册时才这么做。当直接在浏览器中键入URL时，它可能会有副作用。

很多常用的路径扩展名默认在白名单中。此外，REST API调用通常不是直接在浏览器中用作RUL。使用自定义HttpMessageConverter实现的应用可以为内容协商显式注册文件扩展名，响应头Content-Disposition不会被添加到这样的扩展名中。参见 [18.16.6节“内容协商”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-content-negotiation)

|  |
| --- |
| [Note] |
| 这最初在[CVE-2015-5211](http://pivotal.io/security/cve-2015-5211)工作报告中引进。下面是这份报告的补充建议：   * 编写而非避开JSON响应。这也是OWASP XSS中的建议。怎样用Spring这样做的例子见[spring-jackson-owasp](https://github.com/rwinch/spring-jackson-owasp)。 * 配置后缀模式匹配为关闭的，或只限于使用显式注册的后缀。 * 配置内容协商时，设置属性“useJaf”和“ignoreUnknownPathExtensions”为false，这样的话，如果URL中包含未知的扩展名，就会返回一个406错误。注意，如果URL可能会有一个点在结尾的话，这不是一个好的选择。 * Add X-Content-Type-Options: nosniff header to responses. Spring Security 4 does this by default. |

##### 矩阵变量

根据URI规范[RFC 3986](http://tools.ietf.org/html/rfc3986#section-3.3)中URL的定义，路径片段中可以可以包含键值对。规范中没对对应的术语。一般“URL路径参数”可以被应用，尽管更加独特的[“矩阵URI”](http://www.w3.org/DesignIssues/MatrixURIs.html)也经常被使用并且相当有名。在Spring MVC它被成为矩阵变量。

矩阵变量可以出现在任何路径片段中，每一个矩阵变量都用分号（;）隔开。比如“/cars;color=red;year=2012”。多个值可以用逗号隔开，比如“color=red,green,blue”，或者分开写“color=red;color=green;color=blue”。

如果你希望一个URL包含矩阵变量，那么请求映射模式必须用URI模板来表示这些矩阵变量。这样的话，不管矩阵变量顺序如何，都能够保证请求可以正确的匹配。

下面是一个提取矩阵变量“q”的例子：

// GET /pets/42;q=11;r=22

*@GetMapping("/pets/{petId}")*

public void findPet(*@PathVariable* String petId, *@MatrixVariable* int q)

{

// petId == 42

// q == 11

}

因为所有的路径片段都可以包含矩阵变量，所以在一些情况下，你需要格外注意这些变量要出现的位置：

// GET /owners/42;q=11/pets/21;q=22

*@GetMapping("/owners/{ownerId}/pets/{petId}")*

public void findPet(

*@MatrixVariable(name="q", pathVar="ownerId")* int q1,

*@MatrixVariable(name="q", pathVar="petId")* int q2)

{

// q1 == 11

// q2 == 22

}

矩阵变量可以定义为可选的，并为它指定默认值：

// GET /pets/42

*@GetMapping("/pets/{petId}")*

public void findPet(*@MatrixVariable(required=false, defaultValue="1")* int q)

{

// q == 1

}

所有的矩阵变量都可以放到一个Map中：

// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

*@GetMapping("/owners/{ownerId}/pets/{petId}")*

public void findPet(

*@MatrixVariable* MultiValueMap<String, String> matrixVars,

*@MatrixVariable(pathVar="petId"")* MultiValueMap<String, String> petMatrixVars)

{

// matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]

// petMatrixVars: ["q" : 11, "s" : 23]

}

注意，在使用矩阵变量的时候，你必须设置RequestMappingHandlerMapping的removeSemicolonContent属性为false。默认是true。

|  |  |
| --- | --- |
| [Note] | Spring MVC的Java配置和命名空间都提供了启用矩阵变量的选项  如果你使用Java配置，[Advanced Customizations with MVC Java Config](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-advanced-java)一节描述了怎样自定义RequestMappingHandlerMapping。  在MVC的命名空间中，<mvc:annotation-driven>元素有一个enable-matrix-variables属性应该被设置为true，默认是false：  <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  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.xsd  http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd">  <mvc:annotation-driven enable-matrix-variables="true"/>  </beans> |

##### 可消费的媒体类型（Consumable Media Types）

你可以通过指定一个可消费媒体类型来细化主要映射。只有在请求头Content-Type匹配到指定的媒体类型时才匹配请求。例如：

@PostMapping(path = "/pets", **consumes = "application/json"**)

public void addPet(*@RequestBody* Pet pet, Model model)

{

// implementation omitted

}

可消费媒体类型可以使用非操作符取反，比如“!text/plain”匹配所有的Content-Type不是“text/plain”的请求。也可以考虑使用MediaType 提供的常量，比如：APPLICATION\_JSON\_VALUE和APPLICATION\_JSON\_UTF8\_VALUE。

|  |  |
| --- | --- |
| [Note] | *消费*条件可以使用在类型和方法级别上。不像其他的条件，当使用在类型级别上时，方法级别的可消费类型会被覆盖，而不是继承类型级别的条件。 |

##### 可生产的媒体类型（Producible Media Types）

你可以通过指定一个可生产的媒体类型来细化主要映射。只有当请求头匹配其中之一时，才匹配请求。而且，利用生产条件，确保用于生成响应的实际内容类型尊重（？respects）生产条件中指定的媒体类型。例如：

@GetMapping(path = "/pets/{petId}", **produces = MediaType.APPLICATION\_JSON\_UTF8\_VALUE**)

*@ResponseBody*

public Pet getPet(*@PathVariable* String petId, Model model)

{

// implementation omitted

}

|  |  |
| --- | --- |
| [Note] | 注意到指定到*生产*条件的媒体类型可以顺便指定一个字符集。For example, in the code snippet above we specify the same media type than the default one configured in MappingJackson2HttpMessageConverter, including the UTF-8 charset. |

Just like with *consumes*, producible media type expressions can be negated as in !text/plain to match to all requests other than those with an Accept header value of text/plain. Also consider using constants provided in MediaType such as APPLICATION\_JSON\_VALUE and APPLICATION\_JSON\_UTF8\_VALUE.

|  |  |
| --- | --- |
| [Note] | The *produces* condition is supported on the type and on the method level. Unlike most other conditions, when used at the type level, method-level producible types override rather than extend type-level producible types. |

##### 请求参数和头部值

你可以使用诸如“myParam”、“!myParam”或“myParam=myValue”的请求参数条件来细化请求匹配。前两个示例用于测试存在或者不存在请求参数的情况，第三个用于测试指定参数值的情况。这个例子指定了请求参数值：

*@Controller*

*@RequestMapping("/owners/{ownerId}")*

public class RelativePathUriTemplateController

{

@GetMapping(path = "/pets/{petId}", **params = "myParam=myValue"**)

public void findPet(*@PathVariable* String ownerId, *@PathVariable* String petId, Model model)

{

// implementation omitted

}

}

同样，可以测试请求是否存在指定请求头，or to match based on a specific request header value:

*@Controller*

*@RequestMapping("/owners/{ownerId}")*

public class RelativePathUriTemplateController

{

@GetMapping(path = "/pets", **headers = "myHeader=myValue"**)

public void findPet(*@PathVariable* String ownerId, *@PathVariable* String petId, Model model)

{

// implementation omitted

}

}

|  |
| --- |
| [Tip] |
| 尽管你可以使用媒体类型通配符来匹配*Content-Type*和*Accept*值（比如*"content-type=text/\*"*会匹配到*"text/plain"*和*"text/html"*），但是建议使用条件*consumes*和*produces*来代替。因为它们是专为此设计的。 |

##### HTTP HEAD和HTTP OPTIONS

映射到“GET”@RequestMapping方法也隐式地映射到“HEAD”，例如，这样就不需要显式地声明“HEAD”了。一个HTTP HEAD请求的处理过程就像HTTP GET那样，不过它要把请求内容的字节数设置到“Content-Length”中。

@RequestMapping方法内置了HTTP OPTIONS支持。. By default an HTTP OPTIONS request is handled by setting the "Allow" response header to the HTTP methods explicitly declared on all @RequestMapping methods with matching URL patterns. When no HTTP methods are explicitly declared the "Allow" header is set to "GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS". Ideally always declare the HTTP method(s) that an @RequestMapping method is intended to handle, or alternatively use one of the dedicated *composed* @RequestMapping variants (see [the section called “Composed @RequestMapping Variants”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-requestmapping-composed)).

Although not necessary an @RequestMapping method can be mapped to and handle either HTTP HEAD or HTTP OPTIONS, or both.

#### 18.3.3 定义@RequestMapping处理方法

@RequestMapping处理方法可以有非常灵活的签名。支持的方法参数和返回值会在下面描述。大多数参数可以按任意顺序使用，只有BindingResult 参数例外。这将在下面描述。

|  |
| --- |
| [Note] |
| Spring 3.1为@RequestMapping方法引进了一组支持类，分别是RequestMappingHandlerMapping和RequestMappingHandlerAdapter。它们被建议使用，并且它们需要使用Spring MVC 3.1以及以后版本中的新特性。这些支持类默认在MVC的命名空间和Java配置中使用，如果不打算使用，必须进行显式地配置。 |

##### 支持的方法参数类型

下面是支持的方法参数类型：

* （Servlet API 的）请求或者响应对象。可以是任何指定的请求或响应类型，比如ServletRequest或者HttpServletRequest。
* （Servlet API 的）HttpSession类型的Session对象。这种类型的参数强制传入一个相应的回话。所以，这样的参数永远不会为null。

|  |
| --- |
| [Note] |
| 回话访问可能不是线程安全的，尤其是在Servlet环境中。如果允许多个请求同时访问一个回话，应考虑把RequestMappingHandlerAdapter的synchronizeOnSession属性设置为 "true"。. |

* org.springframework.web.context.request.WebRequest或者org.springframework.web.context.request.NativeWebRequest。允许泛型请求参数访问和请求/回话的attribute访问，与本地Servlet/Portlet API解耦。
* 当前请求所在区域的java.util.Locale，通常由指定的区域解析器指定，要在MVC环境中配置LocaleResolver或者LocaleContextResolver。
* 用于当前请求所在时区的java.util.TimeZone (Java 6+)或java.time.ZoneId (Java 8)对象，由LocaleContextResolver决定。
* 访问请求时的java.io.InputStream或java.io.Reader对象。值是由Servlet API暴露出来的未经处理的InputStream或Reader对象。
* 用于生产响应内容的java.io.OutputStream或java.io.Writer对象。值是由Servlet API暴露出来的未经处理的OutputStream或Writer对象。
* 用于HTTP请求方法的org.springframework.http.HttpMethod对象。
* 包含当前被授权用户的java.security.Principal对象。
* 用于访问URI模板变量的，带有@PathVariable注解的参数。见[“URI 模板模式”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-requestmapping-uri-templates)一节。
* 用于访问URI路径片段中的名称-值对的，带有@MatrixVariable注解的变量。见[“矩阵变量”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-matrix-variables)一节.。
* 用于访问指定Servlet请求参数的，带有@RequestParam注解的参数。参数值被转换为生命的方法参数类型。见[“用@RequestParam把请求参数绑定到方法参数”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-requestparam)一节。
* 用于访问指定Servlet请求的HTTP头的，带有@RequestHeader注解的参数。参数值被转换为方法中声明的参数的类型。见[“用@RequestHeader注解映射请求头属性”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-requestheader)一节.
* 用于访问HTTP请求体的，带有@RequestBody注解的参数。参数值通过HttpMessageConverters被转换为方法中声明的参数类型。见[“用@RequestBody注解映射请求体”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-requestbody)一节.
* 用于访问一个"multipart/form-data" 请求内容的，带有@RequestPart注解的参数。见[18.10.5节“编程方式处理来自客户端的文件上传请求”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-multipart-forms-non-browsers) 和[18.10节“Spring的多部分（文件上传）支持”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-multipart).
* 带有@SessionAttribute注解的参数，用于访问已存在的、永久的会话属性（比如用户授权对象），而不是通过@SessionAttributes暂时存储在会话中作为控制器工作流的一部分的模型属性。
* HttpEntity<?>参数，用于访问Servlet请求HTTP头和内容。请求流会通过HttpMessageConverters转换到实体中。见[“使用HttpEntity”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-httpentity)一节.
* java.util.Map或org.springframework.ui.Model或org.springframework.ui.ModelMap对象，用于丰富暴露给Web视图的隐式模式
* org.springframework.web.servlet.mvc.support.RedirectAttributes，用于指定一组精确的属性用于一个重定向，也可以用添加闪存属性（flash attributes）（这些属性被暂时存放在服务器端，以便使请求在重定向之后可以使用）。见 [“传递数据到重定向目标”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-redirecting-passing-data)一节和[18.6节, “使用闪存属性”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-flash-attributes).
* 绑定请求参数到Bean属性（通过set\*）或字段的命令或者表达对象，可以根据@InitBinder方法和/或HandlerAdapter 配置来自定义类型转换。见RequestMappingHandlerAdapter的webBindingInitializer属性。这样的命令对象和它们的校验结果会默认被暴露给模型属性，属性名来自命令类名称，比如来自命令对象的模型属性“orderAddress”的类型是“some.package.OrderAddress”。@ModelAttribute注解可以用在方法参数上来自定义要使用的模型属性名。
* org.springframework.validation.Errors或org.springframework.validation.BindingResult校验结果，用于前面说的命令和表单对象（紧接着这些对象）。
* org.springframework.web.bind.support.SessionStatus状态处理，用于标记表单处理为完成，这会触发清空注解@SessionAttributes在类型级别指定的会话属性。
* org.springframework.web.util.UriComponentsBuilder对象，用于准备一个与当前请求的主机号、端口号、方案（scheme）、上下文路径和Servlet映射的文字部分相关的URL。

Errors或BindingResult参数要紧跟在它要绑定的模型对象后面，因为方法签名可能有多个模型对象，Spring会为它们中的每一个都创建一个 BindingResult实例。因此，下面的例子不会工作。

**无效的BindingResult和@ModelAttribute.的顺序**

*@PostMapping*

public String processSubmit(**@ModelAttribute("pet") Pet pet**, Model model, **BindingResult result**) { ... }

注意，有一个Model参数在Pet和BindingResult之间。为了使它工作，你需要对参数进行重新排序：

*@PostMapping*

public String processSubmit(**@ModelAttribute("pet") Pet pet**, **BindingResult result**, Model model) { ... }

|  |
| --- |
| [Note] |
| JDK 1.8的java.util.Optional也被支持，当带有注解的方法参数类型有一个required属性(例如 @RequestParam, @RequestHeader等)。这种情况下使用java.util.Optional相当于有一个required=false. |

##### 支持的方法返回类型

支持下面的返回类型：

* ModelAndView对象，它包含的model中含有命令对象和带有@ModelAttribute注解的数据访问方法的结果
* Model object, with the view name implicitly determined through a RequestToViewNameTranslator and the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
* A Map object for exposing a model, with the view name implicitly determined through a RequestToViewNameTranslator and the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
* A View object, with the model implicitly determined through command objects and @ModelAttribute annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).
* A String value that is interpreted as the logical view name, with the model implicitly determined through command objects and @ModelAttribute annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).
* void if the method handles the response itself (by writing the response content directly, declaring an argument of type ServletResponse / HttpServletResponse for that purpose) or if the view name is supposed to be implicitly determined through a RequestToViewNameTranslator (not declaring a response argument in the handler method signature).
* If the method is annotated with @ResponseBody, the return type is written to the response HTTP body. The return value will be converted to the declared method argument type using HttpMessageConverters. See [the section called “Mapping the response body with the @ResponseBody annotation”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-responsebody).
* An HttpEntity<?> or ResponseEntity<?> object to provide access to the Servlet response HTTP headers and contents. The entity body will be converted to the response stream using HttpMessageConverters. See [the section called “Using HttpEntity”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-httpentity).
* An HttpHeaders object to return a response with no body.
* A Callable<?> can be returned when the application wants to produce the return value asynchronously in a thread managed by Spring MVC.
* A DeferredResult<?> can be returned when the application wants to produce the return value from a thread of its own choosing.
* A ListenableFuture<?> can be returned when the application wants to produce the return value from a thread of its own choosing.
* A ResponseBodyEmitter can be returned to write multiple objects to the response asynchronously; also supported as the body within a ResponseEntity.
* An SseEmitter can be returned to write Server-Sent Events to the response asynchronously; also supported as the body within a ResponseEntity.
* A StreamingResponseBody can be returned to write to the response OutputStream asynchronously; also supported as the body within a ResponseEntity.
* Any other return type is considered to be a single model attribute to be exposed to the view, using the attribute name specified through @ModelAttribute at the method level (or the default attribute name based on the return type class name). The model is implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.

##### 使用@RequestParam绑定请求参数到方法参数

使用@RequestParam注解绑定请求参数到你的控制器方法中的参数。下面的代码段展示了这种用法：

*@Controller*

*@RequestMapping("/pets")*

*@SessionAttributes("pet")*

public class EditPetForm

{

*@GetMapping*

public String setupForm(**@RequestParam("petId") int petId**, ModelMap model)

{

Pet pet = this.clinic.loadPet(petId);

model.addAttribute("pet", pet);

return "petForm";

}

// ...

}

默认情况下，使用这个注解的参数是必须的，但是你可以通过设置@RequestParam的required属性为false来指定参数是可选的（如： @RequestParam(path="id", required=false)）。

如果目标方法的类型参数不是String类型的，将自动应用类型转换。见 [“方法参数和类型转换”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-typeconversion)一节.

当一个@RequestParam注解用在一个Map<String, String>或者MultiValueMap<String, String>类型的参数上是，这个map将包含所有请求参数。

##### 使用@RequestBody注解映射请求体

方法参数注解@RequestBody指定一个方法参数应该被绑定到HTTP请求体的值。例如：

*@PutMapping("/something")*

public void handle(*@RequestBody* String body, Writer writer) throws IOException

{

writer.write(body);

}

通过使用一个HttpMessageConverter，你能够把请求体转换为方法参数。HttpMessageConverter负责把HTTP请求消息转换为一个对象，也把一个对象转换为HTTP请求体。RequestMappingHandlerAdapter支持@RequestBody注解使用下面的默认的HttpMessageConverters:

* ByteArrayHttpMessageConverter转换字节数组
* StringHttpMessageConverter转换字符串
* FormHttpMessageConverter在表单数据和MultiValueMap<String, String>对象之间转换.
* SourceHttpMessageConverter在请求体和javax.xml.transform.Source之间转换.

更多关于这些转换器的信息，见消息转换器。也要注意，如果使用MVC命名空间或者MVCJava配置，一个更大范围的消息转换器被默认注册。更多信息见[18.16.1节“使用MVC Java配置或者MVC XML命名空间”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-enable)

如果你打算读写XML，你会需要配置一个带有指定Marshaller和Unmarshaller（在包org.springframework.oxm中）实现的MarshallingHttpMessageConverter。虽然下面的例子展示了怎让直接在你的配置中做，但是如果你的应用程序通过MVC命名空间或者MVC Java配置，你要参考[18.16.1节“使用MVC Java配置或者MVC XML命名空间”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-enable)

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">

<property name="messageConverters">

<util:list id="beanList">

<ref bean="stringHttpMessageConverter"/>

<ref bean="marshallingHttpMessageConverter"/>

</util:list>

</property>

</bean>

<bean id="stringHttpMessageConverter" class="org.springframework.http.converter.StringHttpMessageConverter"/>

<bean id="marshallingHttpMessageConverter" class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">

<property name="marshaller" ref="castorMarshaller"/>

<property name="unmarshaller" ref="castorMarshaller"/>

</bean>

<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>

一个@RequestBody方法参数可以带有注解@Valid，此时它会被配置好的Validator 实例校验。当使用MVC命名空间或者MVC Java配置时，一个JSR-303校验器会自动配置进来，如果类路径中有一个可用的JSR-303实现的话。

就像@ModelAttribute参数一样，一个Errors参数可以用于检查错误。如果没有声明这样的参数，一个MethodArgumentNotValidException异常将会抛出。这个异常会在DefaultHandlerExceptionResolver中处理，并返回一个400错误到客户端。

|  |
| --- |
| [Note] |
| 通过MVC命名空间或者Java配置消息转换器和校验器的信息参见[18.16.1节“使用MVC Java配置或者MVC XML命名空间”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-enable)。 |

##### 使用注解@ResponseBody映射响应体

注解@ResponseBody和@RequestBody很像。这个注解可以放到一个方法中，指示返回类型应给直接被写进HTTP响应体中（而不是放到一个Model中、也不是翻译为一个视图名）。例如：

*@GetMapping("/something")*

*@ResponseBody*

public String helloWorld() { return "Hello World"; }

上面的示例会返回文本“Hello World”，并把它写到HTTP响应流中。

就像@RequestBody，Spring使用HttpMessageConverter把返回的对象转换到请求体中。更多关于这些转换器的消息，参见[消息转换器](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#rest-message-conversion).

##### 使用注解@RestController创建REST控制器

一个非常常见的场景是让控制器实现REST API，因此只服务于JSON、XML 或者自定义的MediaType内容。为了方便，避免在所有的@RequestMapping方法上加上@ResponseBody注解，你可以用@RestController注解你的控制器类.

[@RestController](http://docs.spring.io/spring-framework/docs/5.0.0.M1/javadoc-api/org/springframework/web/bind/annotation/RestController.html) 是一个模板注解，它合并了@ResponseBody和@Controller。不仅如此，它还为你的控制器赋予了更多的意义，在未来的发行版本中也将包含更多语义。

就像常规的@Controllers，@RestController可以由@ControllerAdvice或者@RestControllerAdvice Bean协助。详情见[用@ControllerAdvice和@RestControllerAdvice通知控制器”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-controller-advice)一节

##### 使用HttpEntity

HttpEntity和@RequestBody和@ResponseBody很像。出了能够访问请求和响应体，HttpEntity（和子类ResponseEntity）也能访问请求（和响应）头，就像这样：

*@RequestMapping("/something")*

public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity) throws UnsupportedEncodingException

{

String requestHeader = requestEntity.getHeaders().getFirst("MyRequestHeader");

byte[] requestBody = requestEntity.getBody();

// do something with request header and body

HttpHeaders responseHeaders = new HttpHeaders();

responseHeaders.set("MyResponseHeader", "MyValue");

return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);

}

上面的例子获取MyRequestHeader请求头的值，读取请求体为字节数组。还把MyResponseHeader添加到响应中，把Hello World写到响应流中，并设置响应状态码为201（创建）

就像@RequestBody和@ResponseBody一样，Spring使用HttpMessageConverter在请求和响应流只见进行转换。更多关于这些转换器的消息，参见[消息转换器](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#rest-message-conversion).

##### 在方法上使用@ModelAttribute注解

注解@ModelAttribute可以用在方法或者方法参数上。这一节解释它用在方法上，下一节解释把它用在方法参数上。

用在方法上的@ModelAttribute指示那个方法的目的是添加一个或多个模型属性（model attributes）。这样的方法和@RequestMapping方法支持的参数类型相同，但是不能直接映射到请求。在控制器中的@ModelAttribute方法的调用时机是在同一个控制器中的@RequestMapping方法之前。下面是几个例子：

// 添加一个属性

// 这个方法的返回值被添加到名为 “account” 的模型下

// 你可以自定义这个名字，像这样 @ModelAttribute("myAccount")

@ModelAttribute

public Account addAccount(@RequestParam String number) { return accountManager.findAccount(number); }

// 添加多个属性

@ModelAttribute

public void populateModel(@RequestParam String number, Model model)

{

model.addAttribute(accountManager.findAccount(number));

// add more ...

}

@ModelAttribute方法用于构建带有常用属性的模型，比如用状态或者宠物类型填充一个下拉框，或者取回一个命令对象如Account，把它用作HTML表单数据。后一种情况在下一节会进行更深入的讨论。

注意，有两种形式的@ModelAttribute方法。第一种，方法通过返回属性来隐式地添加属性。第二种，方法接受一个Model参数，并把任何数量的属性添加给它。你可以根据需要选择其中的任何一种。

一个控制器可以有任何数量的@ModelAttribute方法。所有的这些方法都在同一个控制器中的@RequestMapping方法之前调用。

@ModelAttribute方法也可以定义在一个带有@ControllerAdvice注解的类中，这些方法应用于很多控制器。详情参见[“使用@ControllerAdvice和@RestControllerAdvice通知控制”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-controller-advice)一节

|  |
| --- |
| [Tip] |
| 在不显式指定属性名的时候发生了什么？这种情况下，被赋值的模型属性的缺省名称是基于其类型的。比如，如果一个方法返回一个Account类型的对象的话，那么缺省名就是“account”。你可以通过注解@ModelAttribute的值来改变。如果直接添加属性到模型的话，请使用合适的方法addAttribute(..) 的重载。 |

注解@ModelAttribute也可以用在@RequestMapping方法上。这种情况下，@RequestMapping方法的返回值被拦截为一个模型属性，而不是一个视图名。关于视图，参见[18.13.3节“视图——RequestToViewNameTranslator”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-coc-r2vnt).

##### 在方法参数上使用@ModelAttribute注解

上节说过，@ModelAttribute可以用在方法或方法参数上。本节解释它用在参数上的情形。

一个用在方法参数上的@ModelAttribute注解指示了参数应该从模型中获取。如果模型中不存在，参数会首先被实例化，然后添加到模型中。一旦模型中存在，这个参数的字段会被所有的名字匹配的请求参数所填充。这在Spring MVC中叫做数据绑定，它能够把你从要对每一个字段进行类型转换的繁重体力劳动中解救出来，是非常有用的机制。

*@PostMapping("/owners/{ownerId}/pets/{petId}/edit")*

public String processSubmit(**@ModelAttribute Pet pet**) { }

上面给出的例子的Pet实例来自哪里？这里有一个可选项：

* 它可能已经存在与模型中了，因为使用了@SessionAttributes — 见[“使用@SessionAttributes存储模型属性到HTTP回话中”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-sessionattrib)一节.
* 它可能已经存在于模型中了，因为同一个控制器中的@ModelAttribute方法，就像上一节中解释的那样。
* 它可能是从URI模板变量和类型转换器中获取的（下面会详细解释）。
* 它可能是使用默认构造器初始化的。

@ModelAttribute方法是一种常用的从数据库中获取属性的方式，可以通过使用@SessionAttributes注解把这种属性在各个请求之间共享。在一些情况下，可以很方便的通过使用URI模板变量和类型转换器来获取这些属性。下面是一个例子：

*@PutMapping("/accounts/{account}")*

public String save(*@ModelAttribute("account")* Account account)

{

// ...

}

在这个例子中，模型属性名（“account”）匹配URI模板变量名。如果你注册的一个Converter<String, Account>可以把字符串account值转换为一个Account，那么上面的例子即使不需要@ModelAttribute也可以正常工作。

下一步是数据绑定。WebDataBinder类匹配请求参数名称——包括请求字符串参数和表单字段——到属性字段名。在必须的类型转换（从字符串到目标类型字段）之后，匹配的字段收集好了。数据绑定和校验见[第五章-检验、数据绑定和类型转换](#_5. 校验、数据绑定和类型转换)。为控制器自定义数据绑定过程见 [“自定义WebDataBinder初始化”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-webdatabinder)一节。

在数据绑定之后，可能会出现一些错误，比如缺少必须字段或者类型转换错误。为了检查这些错误，你可以紧跟在@ModelAttribute参数后面添加一个BindingResult参数：

*@PostMapping("/owners/{ownerId}/pets/{petId}/edit")*

public String processSubmit(**@ModelAttribute("pet") Pet pet**, BindingResult result)

{

if (result.hasErrors()) { return "petForm"; }

// ...

}

有了BindingResult之后，你就可以检查在提交同一个表单时有没有发现错误，这些错误可以通过Spring的<errors>表单标签来显示。

注意到在一些情况下，在model中不使用数据绑定来访问一个属性是很有用的。这种情况下，你可以把Model注入到控制器中，或者在注解上使用绑定标记：

*@ModelAttribute*

public AccountForm setUpForm() { return new AccountForm(); }

*@ModelAttribute*

public Account findAccount(*@PathVariable* String accountId) { return accountRepository.findOne(accountId); }

*@PostMapping("update")*

public String update(*@Valid* AccountUpdateForm form, BindingResult result,

**@ModelAttribute(binding=false)** Account account) { /\* omitted \*/ }

除了数据绑定，你也可以使用你自己定义的、用来传递BindingResult（用于记录数据绑定错误）的校验器调用校验这允许数据绑定和验证的错误积累在一个地方，随后报告给用户：

*@PostMapping("/owners/{ownerId}/pets/{petId}/edit")*

public String processSubmit(**@ModelAttribute("pet") Pet pet**, BindingResult result)

{

new PetValidator().validate(pet, result);

if (result.hasErrors()) return "petForm";

// ...

}

或者你可以自动调用校验，通过添加JSR-303的@Valid注解：

*@PostMapping("/owners/{ownerId}/pets/{petId}/edit")*

public String processSubmit(**@Valid @ModelAttribute("pet") Pet pet**, BindingResult result)

{

if (result.hasErrors()) return "petForm";

// ...

}

关于如何配置和使用校验器，详见[5.8节“Spring校验”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#validation-beanvalidation)和[第五章 校验、数据绑定和类型转换](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#validation)。

##### 使用@SessionAttributes存储模型属性到请求之间的HTTP会话中

类型级别的@SessionAttributes注解声明会话属性会被指定的处理方法使用。This will typically list the names of model attributes or types of model attributes which should be transparently stored in the session or some conversational storage, serving as form-backing beans between subsequent requests.

下面的代码片段展示了这个注解的用法，指定模型属性名：

*@Controller*

*@RequestMapping("/editPet.do")*

**@SessionAttributes("pet")**

public class EditPetForm { /\* omitted \*/ }

##### 使用@SessionAttribute来访问预先存在的全局会话属性

如果你需要访问预先存在的、以全局方式管理的会话属性的话，比如在控制器之外（比如通过过滤器）可能或不可能存在在一个方法参数上使用注解@SessionAttribute：

*@RequestMapping("/")*

public String handle(**@SessionAttribute** User user)

{

// ...

}

为了使用这些需要添加或移除会话属性的情况，考虑注入org.springframework.web.context.request.WebRequest或javax.servlet.http.HttpSession 到一个控制器方法中。

对于暂存在会话中的用作控制器工作流一部分的模型属性，要像[“使用@SessionAttributes存储模型属性到请求共享的HTTP会话”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-sessionattrib)一节中描述的那样使用SessionAttributes。

##### 使用@RequestAttribute访问请求属性

就像@SessionAttribute一样，注解@RequestAttribute可以被用于访问由过滤器或拦截器创建的、预先存在的请求属性：

*@RequestMapping("/")*

public String handle(**@RequestAttribute** Client client)

{

// ...

}

##### 处理“application/x-www-form-urlencoded”数据

前面几节中描述了使用@ModelAttribute支持处理来自浏览器客户端的表单提交请求。这些注解也被推荐用于处理来自非浏览器客户端的请求。但是在处理HTTP PUT请求时，有一处显著不同。浏览器可以通过HTTP GET或者HTTP POST提交表单数据。非浏览器还可以通过HTTP PUT来提交表单。这时会有些棘手，因为Servlet标准要求ServletRequest.getParameter\*()家族的方法，仅用HTTP POST来支持表单字段访问，而不是HTTP PUT。

为了支持PUT和PATCH请求，spring-web模块提供了一个名为HttpPutFormContentFilter的过滤器，可以把它配置在web.xml文件中：

<filter>

<filter-name>httpPutFormFilter</filter-name>

<filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>httpPutFormFilter</filter-name>

<servlet-name>dispatcherServlet</servlet-name>

</filter-mapping>

<servlet>

<servlet-name>dispatcherServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

上面的过滤器拦截内容类型（content type）为application/x-www-form-urlencoded的HTTP PUT和PATCH请求，从请求体中读取表单数据，并包装ServletRequest（为了让表单数据可以通过ServletRequest.getParameter\*()家族方法访问）。

|  |
| --- |
| [Note] |
| 就像HttpPutFormContentFilter消费的请求体那样，它不应该为依赖于其他处理application/x-www-form-urlencoded的转换器的PUT或者PATCH URL配置。这包括@RequestBody MultiValueMap<String, String>和HttpEntity<MultiValueMap<String, String>>。 |

##### 用@CookieValue注解映射cookie值

注解@CookieValue允许一个方法参数允许把一个方法参数绑定到一个HTTP cookie值上。我们来考虑下下面的一个从HTTP请求中接受到的cookie：

JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84

下面的代码示例演示了怎样获取JSESSIONID cookie的值：

*@RequestMapping("/displayHeaderInfo.do")*

public void displayHeaderInfo(**@CookieValue("JSESSIONID")** String cookie)

{

//...

}

如果目标方法参数不是字符串，那么就会自动进行类型转换。见 [“方法参数和类型转换”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-typeconversion)一节。

这个注解支持Servlet和Portlet环境中的处理方法。

##### 用@RequestHeader 映射请求头属性

注解@RequestHeader允许把一个请求头绑定到方法参数上。

这是一个请求头示例：

Host localhost:8080

Accept text/html,application/xhtml+xml,application/xml;q=0.9

Accept-Language fr,en-gb;q=0.7,en;q=0.3

Accept-Encoding gzip,deflate

Accept-Charset ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive 300

下面的示例演示了怎样获取请求头Accept-Encoding和Keep-Alive的值：

*@RequestMapping("/displayHeaderInfo.do")*

public void displayHeaderInfo(**@RequestHeader("Accept-Encoding")** String encoding,

**@RequestHeader("Keep-Alive")** long keepAlive)

{

//...

}

如果目标方法参数不是字符串，那么就会自动进行类型转换。见 [“方法参数和类型转换”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-typeconversion)一节。

当注解@RequestHeader用在一个Map<String, String>、MultiValueMap<String, String>或者HttpHeaders参数上的时候，这个map就是收集所有请求头的值。

|  |
| --- |
| [Tip] |
| 内建的类型转换支持，可以把一个用逗号隔开的字符串，转换为一个字符串或者其他类型的数组或者容器（collection）。例如，一个带有@RequestHeader("Accept")的方法参数可以是一个字符串，也可以是一个String[]或者List<String>。 |

这个注解支持Servlet和Portlet环境中的处理方法。

##### 方法参数和类型转换

从请求中（包括请求参数、路径变量、请求头和cookie）提取的基于字符串的值，可能需要转化为绑定的目标方法参数和字段的类型。如果目标类型并不是String，Spring会自动转换为合适的类型。所有的简单类型，如int, long, Date等，都是支持的。你还可以自定义转换过程，通过一个WebDataBinder（见[“自定义WebDataBinder初始化”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-webdatabinder)一节），或者通过注册FormattingConversionService注册格式化器（见[5.6节“Spring字段格式化”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#format)）。

##### 自定义WebDataBinder初始化

为了自定义基于Spring的WebDataBinder的请求参数和PropertyEditors的绑定，你可以在你的控制器中使用带有@InitBinder注解的方法， @InitBinder方法在一个@ControllerAdvice类内部，或者提供一个自定义的WebBindingInitializer。详情见[“用@ControllerAdvice和@RestControllerAdvice通知控制器”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-controller-advice)一节。

###### 用@InitBinder自定义数据绑定

用@InitBinder注解的控制器方法，允许你直接在你的控制器类中配置Web数据绑定。@InitBinder标记初始化WebDataBinder的方法，WebDataBinder被用于填充被注解的处理方法的命令和表单对象参数。

这些初始化绑定器（Init-binder）方法支持@RequestMapping方法支持的所有参数，处理命令/表单对象以及相关的校验结果对象。初始化绑定器方法必须不带返回值，所以它们通常被声明为void的。典型的参数包括WebDataBinder和WebRequest或者java.util.Locale，允许用代码方式注册特定上下文的编辑器（context-specific editors）。

下面的例子演示了使用@InitBinder为所有的java.util.Date表单属性配置一个CustomDateEditor：

*@Controller*

public class MyFormController

{

**@InitBinder**

protected void initBinder(WebDataBinder binder)

{

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

dateFormat.setLenient(false);

binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));

}

// ...

}

相对地，从Spring 4.2来事，考虑使用addCustomFormatter来指定Formatter实现以代替PropertyEditor实例。This is particularly useful if you happen to have a Formatter-based setup in a shared FormattingConversionService as well, with the same approach to be reused for controller-specific tweaking of the binding rules.

*@Controller*

public class MyFormController

{

**@InitBinder**

protected void initBinder(WebDataBinder binder)

{

binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));

}

// ...

}

###### 配置一个自定义的WebBindingInitializer

为了表达（externalize）数据绑定初始化，你可以提供一个自定义的WebBindingInitializer接口实现，然后你可以通过为RequestMappingHandlerAdapter来提供一个自定义Bean配置来启动WebBindingInitializer，所以要重写默认配置。

下面的例子来自PetClinic应用程序，展示了一个配置，使用一个自定义WebBindingInitializer接口实现——org.springframework.samples.petclinic.web.ClinicBindingInitializer，它配置的PropertyEditors需要几个控制器。

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">

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

<property name="webBindingInitializer">

<bean class="org.springframework.samples.petclinic.web.ClinicBindingInitializer"/>

</property>

</bean>

@InitBinder方法也可以定义在一个带有@ControllerAdvice注解的类中，在这种情况下，它们用于匹配控制器。这提供了一个使用WebBindingInitializer的代替方法。详情见[“使用@ControllerAdvice和@RestControllerAdvice通知控制器一节”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-controller-advice)。

##### 使用@ControllerAdvice和@RestControllerAdvice通知控制器

注解@ControllerAdvice是一个组件注解（component annotation），它允许实现类通过类路径扫描被自动检测到。当使用MVC命名空间或者MVC Java配置时自动启用。

带有@ControllerAdvice注解的类可以包含带有@ExceptionHandler、@InitBinder和@ModelAttribute注解的方法，and these methods will apply to @RequestMapping methods across all controller hierarchies as opposed to the controller hierarchy within which they are declared.

@RestControllerAdvice is an alternative where @ExceptionHandler methods assume @ResponseBody semantics by default.

Both @ControllerAdvice and @RestControllerAdvice can target a subset of controllers:

// Target all Controllers annotated with @RestController

*@ControllerAdvice(annotations = RestController.class)*

public class AnnotationAdvice {}

// Target all Controllers within specific packages

*@ControllerAdvice("org.example.controllers")*

public class BasePackageAdvice {}

// Target all Controllers assignable to specific classes

*@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.class})*

public class AssignableTypesAdvice {}

Check out the [@ControllerAdvice documentation](http://docs.spring.io/spring-framework/docs/5.0.0.M1/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html) for more details.

**Jackson Serialization View Support**

It can sometimes be useful to filter contextually the object that will be serialized to the HTTP response body. In order to provide such capability, Spring MVC has built-in support for rendering with [Jackson’s Serialization Views](http://wiki.fasterxml.com/JacksonJsonViews).

To use it with an @ResponseBody controller method or controller methods that return ResponseEntity, simply add the @JsonView annotation with a class argument specifying the view class or interface to be used:

*@RestController*

public class UserController {

*@GetMapping("/user")*

*@JsonView(User.WithoutPasswordView.class)*

public User getUser() {

return new User("eric", "7!jd#h23");

}

}

public class User {

public interface WithoutPasswordView {};

public interface WithPasswordView extends WithoutPasswordView {};

private String username;

private String password;

public User() {

}

public User(String username, String password) {

this.username = username;

this.password = password;

}

*@JsonView(WithoutPasswordView.class)*

public String getUsername() {

return this.username;

}

*@JsonView(WithPasswordView.class)*

public String getPassword() {

return this.password;

}

}

|  |
| --- |
| [Note] |
| Note that despite @JsonView allowing for more than one class to be specified, the use on a controller method is only supported with exactly one class argument. Consider the use of a composite interface if you need to enable multiple views. |

For controllers relying on view resolution, simply add the serialization view class to the model:

*@Controller*

public class UserController extends AbstractController {

*@GetMapping("/user")*

public String getUser(Model model) {

model.addAttribute("user", new User("eric", "7!jd#h23"));

model.addAttribute(JsonView.class.getName(), User.WithoutPasswordView.class);

return "userView";

}

}

**Jackson JSONP Support**

In order to enable [JSONP](http://en.wikipedia.org/wiki/JSONP) support for @ResponseBody and ResponseEntity methods, declare an @ControllerAdvice bean that extends AbstractJsonpResponseBodyAdvice as shown below where the constructor argument indicates the JSONP query parameter name(s):

*@ControllerAdvice*

public class JsonpAdvice extends AbstractJsonpResponseBodyAdvice {

public JsonpAdvice() {

super("callback");

}

}

For controllers relying on view resolution, JSONP is automatically enabled when the request has a query parameter named jsonp or callback. Those names can be customized through jsonpParameterNames property.

**18.3.4 Asynchronous Request Processing**

Spring MVC 3.2 introduced Servlet 3 based asynchronous request processing. Instead of returning a value, as usual, a controller method can now return a java.util.concurrent.Callable and produce the return value from a Spring MVC managed thread. Meanwhile the main Servlet container thread is exited and released and allowed to process other requests. Spring MVC invokes the Callable in a separate thread with the help of a TaskExecutor and when the Callable returns, the request is dispatched back to the Servlet container to resume processing using the value returned by the Callable. Here is an example of such a controller method:

*@PostMapping*

public Callable<String> processUpload(final MultipartFile file) {

return new Callable<String>() {

public String call() throws Exception {

// ...

return "someView";

}

};

}

Another option is for the controller method to return an instance of DeferredResult. In this case the return value will also be produced from any thread, i.e. one that is not managed by Spring MVC. For example the result may be produced in response to some external event such as a JMS message, a scheduled task, and so on. Here is an example of such a controller method:

*@RequestMapping("/quotes")*

*@ResponseBody*

public DeferredResult<String> quotes() {

DeferredResult<String> deferredResult = new DeferredResult<String>();

// Save the deferredResult somewhere..

return deferredResult;

}

// In some other thread...

deferredResult.setResult(data);

This may be difficult to understand without any knowledge of the Servlet 3.0 asynchronous request processing features. It would certainly help to read up on that. Here are a few basic facts about the underlying mechanism:

* A ServletRequest can be put in asynchronous mode by calling request.startAsync(). The main effect of doing so is that the Servlet, as well as any Filters, can exit but the response will remain open to allow processing to complete later.
* The call to request.startAsync() returns AsyncContext which can be used for further control over async processing. For example it provides the method dispatch, that is similar to a forward from the Servlet API except it allows an application to resume request processing on a Servlet container thread.
* The ServletRequest provides access to the current DispatcherType that can be used to distinguish between processing the initial request, an async dispatch, a forward, and other dispatcher types.

With the above in mind, the following is the sequence of events for async request processing with a Callable:

* Controller returns a Callable.
* Spring MVC starts asynchronous processing and submits the Callable to a TaskExecutor for processing in a separate thread.
* The DispatcherServlet and all Filter’s exit the Servlet container thread but the response remains open.
* The Callable produces a result and Spring MVC dispatches the request back to the Servlet container to resume processing.
* The DispatcherServlet is invoked again and processing resumes with the asynchronously produced result from the Callable.

The sequence for DeferredResult is very similar except it’s up to the application to produce the asynchronous result from any thread:

* Controller returns a DeferredResult and saves it in some in-memory queue or list where it can be accessed.
* Spring MVC starts async processing.
* The DispatcherServlet and all configured Filter’s exit the request processing thread but the response remains open.
* The application sets the DeferredResult from some thread and Spring MVC dispatches the request back to the Servlet container.
* The DispatcherServlet is invoked again and processing resumes with the asynchronously produced result.

For further background on the motivation for async request processing and when or why to use it please read [this blog post series](https://spring.io/blog/2012/05/07/spring-mvc-3-2-preview-introducing-servlet-3-async-support).

**Exception Handling for Async Requests**

What happens if a Callable returned from a controller method raises an Exception while being executed? The short answer is the same as what happens when a controller method raises an exception. It goes through the regular exception handling mechanism. The longer explanation is that when a Callable raises an Exception Spring MVC dispatches to the Servlet container with the Exception as the result and that leads to resume request processing with the Exception instead of a controller method return value. When using a DeferredResult you have a choice whether to call setResult or setErrorResult with an Exception instance.

**Intercepting Async Requests**

A HandlerInterceptor can also implement AsyncHandlerInterceptor in order to implement the afterConcurrentHandlingStarted callback, which is called instead of postHandle and afterCompletion when asynchronous processing starts.

A HandlerInterceptor can also register a CallableProcessingInterceptor or a DeferredResultProcessingInterceptor in order to integrate more deeply with the lifecycle of an asynchronous request and for example handle a timeout event. See the Javadoc of AsyncHandlerInterceptor for more details.

The DeferredResult type also provides methods such as onTimeout(Runnable) and onCompletion(Runnable). See the Javadoc of DeferredResult for more details.

When using a Callable you can wrap it with an instance of WebAsyncTask which also provides registration methods for timeout and completion.

**HTTP Streaming**

A controller method can use DeferredResult and Callable to produce its return value asynchronously and that can be used to implement techniques such as [long polling](http://spring.io/blog/2012/05/08/spring-mvc-3-2-preview-techniques-for-real-time-updates/) where the server can push an event to the client as soon as possible.

What if you wanted to push multiple events on a single HTTP response? This is a technique related to "Long Polling" that is known as "HTTP Streaming". Spring MVC makes this possible through the ResponseBodyEmitter return value type which can be used to send multiple Objects, instead of one as is normally the case with @ResponseBody, where each Object sent is written to the response with an HttpMessageConverter.

Here is an example of that:

*@RequestMapping("/events")*

public ResponseBodyEmitter handle() {

ResponseBodyEmitter emitter = new ResponseBodyEmitter();

// Save the emitter somewhere..

return emitter;

}

// In some other thread

emitter.send("Hello once");

// and again later on

emitter.send("Hello again");

// and done at some point

emitter.complete();

Note that ResponseBodyEmitter can also be used as the body in a ResponseEntity in order to customize the status and headers of the response.

**HTTP Streaming With Server-Sent Events**

SseEmitter is a sub-class of ResponseBodyEmitter providing support for [Server-Sent Events](http://www.w3.org/TR/eventsource/). Server-sent events is a just another variation on the same "HTTP Streaming" technique except events pushed from the server are formatted according to the W3C Server-Sent Events specification.

Server-Sent Events can be used for their intended purpose, that is to push events from the server to clients. It is quite easy to do in Spring MVC and requires simply returning a value of type SseEmitter.

Note however that Internet Explorer does not support Server-Sent Events and that for more advanced web application messaging scenarios such as online games, collaboration, financial applicatinos, and others it’s better to consider Spring’s WebSocket support that includes SockJS-style WebSocket emulation falling back to a very wide range of browsers (including Internet Explorer) and also higher-level messaging patterns for interacting with clients through a publish-subscribe model within a more messaging-centric architecture. For further background on this see [the following blog post](http://blog.pivotal.io/pivotal/products/websocket-architecture-in-spring-4-0).

**HTTP Streaming Directly To The OutputStream**

ResponseBodyEmitter allows sending events by writing Objects to the response through an HttpMessageConverter. This is probably the most common case, for example when writing JSON data. However sometimes it is useful to bypass message conversion and write directly to the response OutputStream for example for a file download. This can be done with the help of the StreamingResponseBody return value type.

Here is an example of that:

*@RequestMapping("/download")*

public StreamingResponseBody handle() {

return new StreamingResponseBody() {

*@Override*

public void writeTo(OutputStream outputStream) throws IOException {

// write...

}

};

}

Note that StreamingResponseBody can also be used as the body in a ResponseEntity in order to customize the status and headers of the response.

**Configuring Asynchronous Request Processing**

**Servlet Container Configuration**

For applications configured with a web.xml be sure to update to version 3.0:

<web-app xmlns="http://java.sun.com/xml/ns/javaee"

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

http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

version="3.0">

...

</web-app>

Asynchronous support must be enabled on the DispatcherServlet through the <async-supported>true</async-supported> sub-element in web.xml. Additionally any Filter that participates in asyncrequest processing must be configured to support the ASYNC dispatcher type. It should be safe to enable the ASYNC dispatcher type for all filters provided with the Spring Framework since they usually extend OncePerRequestFilter and that has runtime checks for whether the filter needs to be involved in async dispatches or not.

Below is some example web.xml configuration:

<web-app xmlns="http://java.sun.com/xml/ns/javaee"

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

xsi:schemaLocation="

http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

version="3.0">

<filter>

<filter-name>Spring OpenEntityManagerInViewFilter</filter-name>

<filter-class>org.springframework.~.OpenEntityManagerInViewFilter</filter-class>

<async-supported>true</async-supported>

</filter>

<filter-mapping>

<filter-name>Spring OpenEntityManagerInViewFilter</filter-name>

<url-pattern>/\*</url-pattern>

<dispatcher>REQUEST</dispatcher>

<dispatcher>ASYNC</dispatcher>

</filter-mapping>

</web-app>

If using Servlet 3, Java based configuration for example via WebApplicationInitializer, you’ll also need to set the "asyncSupported" flag as well as the ASYNC dispatcher type just like with web.xml. To simplify all this configuration, consider extending AbstractDispatcherServletInitializer, or better AbstractAnnotationConfigDispatcherServletInitializer which automatically set those options and make it very easy to register Filter instances.

**Spring MVC Configuration**

The MVC Java config and the MVC namespace provide options for configuring asynchronous request processing. WebMvcConfigurer has the method configureAsyncSupport while <mvc:annotation-driven> has an <async-support> sub-element.

Those allow you to configure the default timeout value to use for async requests, which if not set depends on the underlying Servlet container (e.g. 10 seconds on Tomcat). You can also configure an AsyncTaskExecutor to use for executing Callable instances returned from controller methods. It is highly recommended to configure this property since by default Spring MVC uses SimpleAsyncTaskExecutor. The MVC Java config and the MVC namespace also allow you to register CallableProcessingInterceptor and DeferredResultProcessingInterceptor instances.

If you need to override the default timeout value for a specific DeferredResult, you can do so by using the appropriate class constructor. Similarly, for a Callable, you can wrap it in a WebAsyncTask and use the appropriate class constructor to customize the timeout value. The class constructor of WebAsyncTask also allows providing an AsyncTaskExecutor.

**18.3.5 Testing Controllers**

The spring-test module offers first class support for testing annotated controllers. See [Section 11.6, “Spring MVC Test Framework”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#spring-mvc-test-framework).

**18.4 Handler mappings**

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. However, do keep in mind that all HandlerMapping classes extending from AbstractHandlerMapping have the following properties that you can use to customize their behavior:

* interceptors List of interceptors to use. HandlerInterceptors are discussed in [Section 18.4.1, “Intercepting requests with a HandlerInterceptor”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-handlermapping-interceptor).
* defaultHandler Default handler to use, when this handler mapping does not result in a matching handler.
* order Based on the value of the order property (see the org.springframework.core.Ordered interface), Spring sorts all handler mappings available in the context and applies the first matching handler.
* alwaysUseFullPath If true , Spring uses the full path within the current Servlet context to find an appropriate handler. If false (the default), the path within the current Servlet mapping is used. For example, if a Servlet is mapped using /testing/\* and the alwaysUseFullPath property is set to true, /testing/viewPage.html is used, whereas if the property is set to false, /viewPage.html is used.
* urlDecode Defaults to true, as of Spring 2.5. If you prefer to compare encoded paths, set this flag to false. However, the HttpServletRequest always exposes the Servlet path in decoded form. Be aware that the Servlet path will not match when compared with encoded paths.

The following example shows how to configure an interceptor:

<beans>

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">

<property name="interceptors">

<bean class="example.MyInterceptor"/>

</property>

</bean>

</beans>

**18.4.1 Intercepting requests with a HandlerInterceptor**

Spring’s handler mapping mechanism includes handler interceptors, which are useful when you want to apply specific functionality to certain requests, for example, checking for a principal.

Interceptors located in the handler mapping must implement HandlerInterceptor from the org.springframework.web.servlet package. This interface defines three methods: preHandle(..) is called *before* the actual handler is executed; postHandle(..) is called *after* the handler is executed; and afterCompletion(..) is called *after the complete request has finished*. These three methods should provide enough flexibility to do all kinds of preprocessing and postprocessing.

The preHandle(..) method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue; when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

Interceptors can be configured using the interceptors property, which is present on all HandlerMapping classes extending from AbstractHandlerMapping. This is shown in the example below:

<beans>

<bean id="handlerMapping"

class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">

<property name="interceptors">

<list>

<ref bean="officeHoursInterceptor"/>

</list>

</property>

</bean>

<bean id="officeHoursInterceptor"

class="samples.TimeBasedAccessInterceptor">

<property name="openingTime" value="9"/>

<property name="closingTime" value="18"/>

</bean>

</beans>

package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

private int openingTime;

private int closingTime;

public void setOpeningTime(int openingTime) {

this.openingTime = openingTime;

}

public void setClosingTime(int closingTime) {

this.closingTime = closingTime;

}

public boolean preHandle(HttpServletRequest request, HttpServletResponse response,

Object handler) throws Exception {

Calendar cal = Calendar.getInstance();

int hour = cal.get(HOUR\_OF\_DAY);

if (openingTime <= hour && hour < closingTime) {

return true;

}

response.sendRedirect("http://host.com/outsideOfficeHours.html");

return false;

}

}

Any request handled by this mapping is intercepted by the TimeBasedAccessInterceptor. If the current time is outside office hours, the user is redirected to a static HTML file that says, for example, you can only access the website during office hours.

|  |
| --- |
| [Note] |
| When using the RequestMappingHandlerMapping the actual handler is an instance of HandlerMethod which identifies the specific controller method that will be invoked. |

As you can see, the Spring adapter class HandlerInterceptorAdapter makes it easier to extend the HandlerInterceptor interface.

|  |
| --- |
| [Tip] |
| In the example above, the configured interceptor will apply to all requests handled with annotated controller methods. If you want to narrow down the URL paths to which an interceptor applies, you can use the MVC namespace or the MVC Java config, or declare bean instances of type MappedInterceptor to do that. See [Section 18.16.1, “Enabling the MVC Java Config or the MVC XML Namespace”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-enable). |

Note that the postHandle method of HandlerInterceptor is not always ideally suited for use with @ResponseBody and ResponseEntity methods. In such cases an HttpMessageConverter writes to and commits the response before postHandle is called which makes it impossible to change the response, for example to add a header. Instead an application can implement ResponseBodyAdvice and either declare it as an @ControllerAdvice bean or configure it directly on RequestMappingHandlerAdapter.

**18.5 Resolving views**

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, FreeMarker templates and XSLT views, for example. See [Chapter 19, *View technologies*](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#view) for a discussion of how to integrate and use a number of disparate view technologies.

The two interfaces that 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.

**18.5.1 Resolving views with the ViewResolver interface**

As discussed in [Section 18.3, “Implementing Controllers”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-controller), all handler methods in the Spring Web MVC controllers must resolve to a logical view name, either explicitly (e.g., by returning a String, View, or ModelAndView) or implicitly (i.e., based on conventions). Views in Spring are addressed by a logical view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. This table lists most of them; a couple of examples follow.

**Table 18.3. View resolvers**

| **ViewResolver** | **Description** |
| --- | --- |
| AbstractCachingViewResolver | Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching. |
| XmlViewResolver | 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 | Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is views.properties. |
| UrlBasedViewResolver | Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings. |
| InternalResourceViewResolver | Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..). See the UrlBasedViewResolver javadocs for details. |
| FreeMarkerViewResolver | Convenient subclass of UrlBasedViewResolver that supports FreeMarkerView and custom subclasses of them. |
| ContentNegotiatingViewResolver | Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header. See [Section 18.5.4, “ContentNegotiatingViewResolver”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-multiple-representations). |

As an example, with JSP as a view technology, you can use the UrlBasedViewResolver. This view resolver translates a view name to a URL and hands the request over to the RequestDispatcher to render the view.

<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>

When returning test as a logical view name, this view resolver forwards the request to the RequestDispatcher that will send the request to /WEB-INF/jsp/test.jsp.

When you combine different view technologies in a web application, you can use the ResourceBundleViewResolver:

<bean id="viewResolver"

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

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

<property name="defaultParentView" value="parentView"/>

</bean>

The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].(class) as the view class and the value of the property [viewname].url as the view url. Examples can be found in the next chapter which covers view technologies. As you can see, you can identify a parent view, from which all views in the properties file "extend". This way you can specify a default view class, for example.

|  |
| --- |
| [Note] |
| Subclasses of AbstractCachingViewResolver cache view instances that they resolve. Caching improves performance of certain view technologies. It’s possible to turn off the cache by setting the cache property to false. Furthermore, if you must refresh a certain view at runtime (for example when a FreeMarker template is modified), you can use the removeFromCache(String viewName, Locale loc) method. |

**18.5.2 Chaining ViewResolvers**

Spring supports multiple view resolvers. Thus you can chain resolvers and, for example, override specific views in certain circumstances. You chain view resolvers by adding more than one resolver to your application context and, if necessary, by setting the order property to specify ordering. Remember, the higher the order property, the later the view resolver is positioned in the chain.

In the following example, the chain of view resolvers consists of two resolvers, an InternalResourceViewResolver, which is always automatically positioned as the last resolver in the chain, and an XmlViewResolver for specifying Excel views. Excel views are not supported by the InternalResourceViewResolver.

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

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

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

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

</bean>

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

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

<property name="location" value="/WEB-INF/views.xml"/>

</bean>

<!-- in views.xml -->

<beans>

<bean name="report" class="org.springframework.example.ReportExcelView"/>

</beans>

If a specific view resolver does not result in a view, Spring examines the context for other view resolvers. If additional view resolvers exist, Spring continues to inspect them until a view is resolved. If no view resolver returns a view, Spring throws a ServletException.

The contract of a view resolver specifies that a view resolver *can* return null to indicate the view could not be found. Not all view resolvers do this, however, because in some cases, the resolver simply cannot detect whether or not the view exists. For example, the InternalResourceViewResolver uses the RequestDispatcher internally, and dispatching is the only way to figure out if a JSP exists, but this action can only execute once. The same holds for the FreeMarkerViewResolver and some others. Check the javadocs of the specific view resolver to see whether it reports non-existing views. Thus, putting an InternalResourceViewResolver in the chain in a place other than the last results in the chain not being fully inspected, because the InternalResourceViewResolver will *always* return a view!

**18.5.3 Redirecting to Views**

As mentioned previously, a controller typically returns a logical view name, which a view resolver resolves to a particular view technology. For view technologies such as JSPs that are processed through the Servlet or JSP engine, this resolution is usually handled through the combination of InternalResourceViewResolver and InternalResourceView, which issues an internal forward or include via the Servlet API’s RequestDispatcher.forward(..) method or RequestDispatcher.include() method. For other view technologies, such as FreeMarker, XSLT, and so on, the view itself writes the content directly to the response stream.

It is sometimes desirable to issue an HTTP redirect back to the client, before the view is rendered. This is desirable, for example, when one controller has been called with POST data, and the response is actually a delegation to another controller (for example on a successful form submission). In this case, a normal internal forward will mean that the other controller will also see the same POST data, which is potentially problematic if it can confuse it with other expected data. Another reason to perform a redirect before displaying the result is to eliminate the possibility of the user submitting the form data multiple times. In this scenario, the browser will first send an initial POST; it will then receive a response to redirect to a different URL; and finally the browser will perform a subsequent GET for the URL named in the redirect response. Thus, from the perspective of the browser, the current page does not reflect the result of a POST but rather of a GET. The end effect is that there is no way the user can accidentally re- POST the same data by performing a refresh. The refresh forces a GET of the result page, not a resend of the initial POST data.

**RedirectView**

One way to force a redirect as the result of a controller response is for the controller to create and return an instance of Spring’s RedirectView. In this case, DispatcherServlet does not use the normal view resolution mechanism. Rather because it has been given the (redirect) view already, the DispatcherServlet simply instructs the view to do its work. The RedirectView in turn calls HttpServletResponse.sendRedirect() to send an HTTP redirect to the client browser.

If you use RedirectView and the view is created by the controller itself, it is recommended that you configure the redirect URL to be injected into the controller so that it is not baked into the controller but configured in the context along with the view names. The [the section called “The redirect: prefix”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-redirecting-redirect-prefix) facilitates this decoupling.

**Passing Data To the Redirect Target**

By default all model attributes are considered to be exposed as URI template variables in the redirect URL. Of the remaining attributes those that are primitive types or collections/arrays of primitive types are automatically appended as query parameters.

Appending primitive type attributes as query parameters may be the desired result if a model instance was prepared specifically for the redirect. However, in annotated controllers the model may contain additional attributes added for rendering purposes (e.g. drop-down field values). To avoid the possibility of having such attributes appear in the URL, an @RequestMapping method can declare an argument of type RedirectAttributes and use it to specify the exact attributes to make available to RedirectView. If the method does redirect, the content of RedirectAttributes is used. Otherwise the content of the model is used.

The RequestMappingHandlerAdapter provides a flag called "ignoreDefaultModelOnRedirect" that can be used to indicate the content of the default Model should never be used if a controller method redirects. Instead the controller method should declare an attribute of type RedirectAttributes or if it doesn’t do so no attributes should be passed on to RedirectView. Both the MVC namespace and the MVC Java config keep this flag set to false in order to maintain backwards compatibility. However, for new applications we recommend setting it to true

Note that URI template variables from the present request are automatically made available when expanding a redirect URL and do not need to be added explicitly neither through Model nor RedirectAttributes. For example:

*@PostMapping("/files/{path}")*

public String upload(...) {

// ...

return "redirect:files/{path}";

}

Another way of passing data to the redirect target is via *Flash Attributes*. Unlike other redirect attributes, flash attributes are saved in the HTTP session (and hence do not appear in the URL). See [Section 18.6, “Using flash attributes”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-flash-attributes) for more information.

**The redirect: prefix**

While the use of RedirectView works fine, if the controller itself creates the RedirectView, there is no avoiding the fact that the controller is aware that a redirection is happening. This is really suboptimal and couples things too tightly. The controller should not really care about how the response gets handled. In general it should operate only in terms of view names that have been injected into it.

The special redirect: prefix allows you to accomplish this. If a view name is returned that has the prefix redirect:, the UrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The net effect is the same as if the controller had returned a RedirectView, but now the controller itself can simply operate in terms of logical view names. A logical view name such as redirect:/myapp/some/resource will redirect relative to the current Servlet context, while a name such as redirect:http://myhost.com/some/arbitrary/path will redirect to an absolute URL.

Note that the controller handler is annotated with the @ResponseStatus, the annotation value takes precedence over the response status set by RedirectView.

**The forward: prefix**

It is also possible to use a special forward: prefix for view names that are ultimately resolved by UrlBasedViewResolver and subclasses. This creates an InternalResourceView (which ultimately does a RequestDispatcher.forward()) around the rest of the view name, which is considered a URL. Therefore, this prefix is not useful with InternalResourceViewResolver and InternalResourceView (for JSPs for example). But the prefix can be helpful when you are primarily using another view technology, but still want to force a forward of a resource to be handled by the Servlet/JSP engine. (Note that you may also chain multiple view resolvers, instead.)

As with the redirect: prefix, if the view name with the forward: prefix is injected into the controller, the controller does not detect that anything special is happening in terms of handling the response.

**18.5.4 ContentNegotiatingViewResolver**

The ContentNegotiatingViewResolver does not resolve views itself but rather delegates to other view resolvers, selecting the view that resembles the representation requested by the client. Two strategies exist for a client to request a representation from the server:

* Use a distinct URI for each resource, typically by using a different file extension in the URI. For example, the URI <http://www.example.com/users/fred.pdf> requests a PDF representation of the user fred, and <http://www.example.com/users/fred.xml> requests an XML representation.
* Use the same URI for the client to locate the resource, but set the Accept HTTP request header to list the [media types](http://en.wikipedia.org/wiki/Internet_media_type) that it understands. For example, an HTTP request for <http://www.example.com/users/fred> with an Accept header set to application/pdf requests a PDF representation of the user fred, while <http://www.example.com/users/fred> with an Accept header set to text/xml requests an XML representation. This strategy is known as [content negotiation](http://en.wikipedia.org/wiki/Content_negotiation).

|  |
| --- |
| [Note] |
| One issue with the Accept header is that it is impossible to set it in a web browser within HTML. For example, in Firefox, it is fixed to:  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  For this reason it is common to see the use of a distinct URI for each representation when developing browser based web applications. |

To support multiple representations of a resource, Spring provides the ContentNegotiatingViewResolver to resolve a view based on the file extension or Accept header of the HTTP request. ContentNegotiatingViewResolver does not perform the view resolution itself but instead delegates to a list of view resolvers that you specify through the bean property ViewResolvers.

The ContentNegotiatingViewResolver selects an appropriate View to handle the request by comparing the request media type(s) with the media type (also known as Content-Type) supported by the View associated with each of its ViewResolvers. The first View in the list that has a compatible Content-Type returns the representation to the client. If a compatible view cannot be supplied by the ViewResolver chain, then the list of views specified through the DefaultViews property will be consulted. This latter option is appropriate for singleton Views that can render an appropriate representation of the current resource regardless of the logical view name. The Accept header may include wild cards, for example text/\*, in which case a View whose Content-Type was text/xml is a compatible match.

To support custom resolution of a view based on a file extension, use a ContentNegotiationManager: see [Section 18.16.6, “Content Negotiation”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-content-negotiation).

Here is an example configuration of a ContentNegotiatingViewResolver:

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

<property name="viewResolvers">

<list>

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

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

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

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

</bean>

</list>

</property>

<property name="defaultViews">

<list>

<bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

</list>

</property>

</bean>

<bean id="content" class="com.foo.samples.rest.SampleContentAtomView"/>

The InternalResourceViewResolver handles the translation of view names and JSP pages, while the BeanNameViewResolver returns a view based on the name of a bean. (See "[Resolving views with the ViewResolver interface](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-viewresolver-resolver)" for more details on how Spring looks up and instantiates a view.) In this example, the content bean is a class that inherits from AbstractAtomFeedView, which returns an Atom RSS feed. For more information on creating an Atom Feed representation, see the section Atom Views.

In the above configuration, if a request is made with an .html extension, the view resolver looks for a view that matches the text/html media type. The InternalResourceViewResolver provides the matching view for text/html. If the request is made with the file extension .atom, the view resolver looks for a view that matches the application/atom+xml media type. This view is provided by the BeanNameViewResolver that maps to the SampleContentAtomView if the view name returned is content. If the request is made with the file extension .json, the MappingJackson2JsonView instance from the DefaultViews list will be selected regardless of the view name. Alternatively, client requests can be made without a file extension but with the Accept header set to the preferred media-type, and the same resolution of request to views would occur.

|  |
| --- |
| [Note] |
| If `ContentNegotiatingViewResolver’s list of ViewResolvers is not configured explicitly, it automatically uses any ViewResolvers defined in the application context. |

The corresponding controller code that returns an Atom RSS feed for a URI of the form <http://localhost/content.atom> or <http://localhost/content> with an Accept header of application/atom+xml is shown below.

*@Controller*

public class ContentController {

private List<SampleContent> contentList = new ArrayList<SampleContent>();

*@GetMapping("/content")*

public ModelAndView getContent() {

ModelAndView mav = new ModelAndView();

mav.setViewName("content");

mav.addObject("sampleContentList", contentList);

return mav;

}

}

**18.6 Using flash attributes**

Flash attributes provide a way for one request to store attributes intended for use in another. This is most commonly needed when redirecting — for example, the *Post/Redirect/Get* pattern. Flash attributes are saved temporarily before the redirect (typically in the session) to be made available to the request after the redirect and removed immediately.

Spring MVC has two main abstractions in support of flash attributes. FlashMap is used to hold flash attributes while FlashMapManager is used to store, retrieve, and manage FlashMap instances.

Flash attribute support is always "on" and does not need to enabled explicitly although if not used, it never causes HTTP session creation. On each request there is an "input" FlashMap with attributes passed from a previous request (if any) and an "output" FlashMap with attributes to save for a subsequent request. Both FlashMap instances are accessible from anywhere in Spring MVC through static methods in RequestContextUtils.

Annotated controllers typically do not need to work with FlashMap directly. Instead an @RequestMapping method can accept an argument of type RedirectAttributes and use it to add flash attributes for a redirect scenario. Flash attributes added via RedirectAttributes are automatically propagated to the "output" FlashMap. Similarly, after the redirect, attributes from the "input" FlashMap are automatically added to the Model of the controller serving the target URL.

**Matching requests to flash attributes**

The concept of flash attributes exists in many other Web frameworks and has proven to be exposed sometimes to concurrency issues. This is because by definition flash attributes are to be stored until the next request. However the very "next" request may not be the intended recipient but another asynchronous request (e.g. polling or resource requests) in which case the flash attributes are removed too early.

To reduce the possibility of such issues, RedirectView automatically "stamps" FlashMap instances with the path and query parameters of the target redirect URL. In turn the default FlashMapManager matches that information to incoming requests when looking up the "input" FlashMap.

This does not eliminate the possibility of a concurrency issue entirely but nevertheless reduces it greatly with information that is already available in the redirect URL. Therefore the use of flash attributes is recommended mainly for redirect scenarios .

**18.7 Building URIs**

Spring MVC provides a mechanism for building and encoding a URI using UriComponentsBuilder and UriComponents.

For example you can expand and encode a URI template string:

UriComponents uriComponents = UriComponentsBuilder.fromUriString(

"http://example.com/hotels/{hotel}/bookings/{booking}").build();

URI uri = uriComponents.expand("42", "21").encode().toUri();

Note that UriComponents is immutable and the expand() and encode() operations return new instances if necessary.

You can also expand and encode using individual URI components:

UriComponents uriComponents = UriComponentsBuilder.newInstance()

.scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}").build()

.expand("42", "21")

.encode();

In a Servlet environment the ServletUriComponentsBuilder sub-class provides static factory methods to copy available URL information from a Servlet requests:

HttpServletRequest request = ...

// Re-use host, scheme, port, path and query string

// Replace the "accountId" query param

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)

.replaceQueryParam("accountId", "{id}").build()

.expand("123")

.encode();

Alternatively, you may choose to copy a subset of the available information up to and including the context path:

// Re-use host, port and context path

// Append "/accounts" to the path

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(request)

.path("/accounts").build()

Or in cases where the DispatcherServlet is mapped by name (e.g. /main/\*), you can also have the literal part of the servlet mapping included:

// Re-use host, port, context path

// Append the literal part of the servlet mapping to the path

// Append "/accounts" to the path

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping(request)

.path("/accounts").build()

**18.7.1 Building URIs to Controllers and methods**

Spring MVC also provides a mechanism for building links to controller methods. For example, given:

*@Controller*

*@RequestMapping("/hotels/{hotel}")*

public class BookingController {

*@GetMapping("/bookings/{booking}")*

public String getBooking(*@PathVariable* Long booking) {

// ...

}

}

You can prepare a link by referring to the method by name:

UriComponents uriComponents = MvcUriComponentsBuilder

.fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

In the above example we provided actual method argument values, in this case the long value 21, to be used as a path variable and inserted into the URL. Furthermore, we provided the value 42 in order to fill in any remaining URI variables such as the "hotel" variable inherited from the type-level request mapping. If the method had more arguments you can supply null for arguments not needed for the URL. In general only @PathVariable and @RequestParam arguments are relevant for constructing the URL.

There are additional ways to use MvcUriComponentsBuilder. For example you can use a technique akin to mock testing through proxies to avoid referring to the controller method by name (the example assumes static import of MvcUriComponentsBuilder.on):

UriComponents uriComponents = MvcUriComponentsBuilder

.fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

The above examples use static methods in MvcUriComponentsBuilder. Internally they rely on ServletUriComponentsBuilder to prepare a base URL from the scheme, host, port, context path and servlet path of the current request. This works well in most cases, however sometimes it may be insufficient. For example you may be outside the context of a request (e.g. a batch process that prepares links) or perhaps you need to insert a path prefix (e.g. a locale prefix that was removed from the request path and needs to be re-inserted into links).

For such cases you can use the static "fromXxx" overloaded methods that accept a UriComponentsBuilder to use base URL. Or you can create an instance of MvcUriComponentsBuilder with a base URL and then use the instance-based "withXxx" methods. For example:

UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path("/en");

MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);

builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

**18.7.2 Building URIs to Controllers and methods from views**

You can also build links to annotated controllers from views such as JSP, Thymeleaf, FreeMarker. This can be done using the fromMappingName method in MvcUriComponentsBuilder which refers to mappings by name.

Every @RequestMapping is assigned a default name based on the capital letters of the class and the full method name. For example, the method getFoo in class FooController is assigned the name "FC#getFoo". This strategy can be replaced or customized by creating an instance of HandlerMethodMappingNamingStrategy and plugging it into your RequestMappingHandlerMapping. The default strategy implementation also looks at the name attribute on @RequestMapping and uses that if present. That means if the default mapping name assigned conflicts with another (e.g. overloaded methods) you can assign a name explicitly on the @RequestMapping.

|  |
| --- |
| [Note] |
| The assigned request mapping names are logged at TRACE level on startup. |

The Spring JSP tag library provides a function called mvcUrl that can be used to prepare links to controller methods based on this mechanism.

For example given:

*@RequestMapping("/people/{id}/addresses")*

public class PersonAddressController {

*@RequestMapping("/{country}")*

public HttpEntity getAddress(*@PathVariable* String country) { ... }

}

You can prepare a link from a JSP as follows:

<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>

...

<a href="${s:mvcUrl('PAC#getAddress').arg(0,'US').buildAndExpand('123')}">Get Address</a>

The above example relies on the mvcUrl JSP function declared in the Spring tag library (i.e. META-INF/spring.tld). For more advanced cases (e.g. a custom base URL as explained in the previous section), it is easy to define your own function, or use a custom tag file, in order to use a specific instance of MvcUriComponentsBuilder with a custom base URL.

**18.8 Using locales**

Most parts of Spring’s architecture support internationalization, just as the Spring web MVC framework does. DispatcherServlet enables you to automatically resolve messages using the client’s locale. This is done with LocaleResolver objects.

When a request comes in, the DispatcherServlet looks for a locale resolver, and if it finds one it tries to use it to set the locale. Using the RequestContext.getLocale() method, you can always retrieve the locale that was resolved by the locale resolver.

In addition to automatic locale resolution, you can also attach an interceptor to the handler mapping (see [Section 18.4.1, “Intercepting requests with a HandlerInterceptor”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-handlermapping-interceptor) for more information on handler mapping interceptors) to change the locale under specific circumstances, for example, based on a parameter in the request.

Locale resolvers and interceptors are defined in the org.springframework.web.servlet.i18n package and are configured in your application context in the normal way. Here is a selection of the locale resolvers included in Spring.

**18.8.1 Obtaining Time Zone Information**

In addition to obtaining the client’s locale, it is often useful to know their time zone. The LocaleContextResolver interface offers an extension to LocaleResolver that allows resolvers to provide a richer LocaleContext, which may include time zone information.

When available, the user’s TimeZone can be obtained using the RequestContext.getTimeZone() method. Time zone information will automatically be used by Date/Time Converter and Formatter objects registered with Spring’s ConversionService.

**18.8.2 AcceptHeaderLocaleResolver**

This locale resolver inspects the accept-language header in the request that was sent by the client (e.g., a web browser). Usually this header field contains the locale of the client’s operating system. *Note that this resolver does not support time zone information.*

**18.8.3 CookieLocaleResolver**

This locale resolver inspects a Cookie that might exist on the client to see if a Locale or TimeZone is specified. If so, it uses the specified details. Using the properties of this locale resolver, you can specify the name of the cookie as well as the maximum age. Find below an example of defining a CookieLocaleResolver.

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

<property name="cookieName" value="clientlanguage"/>

<!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->

<property name="cookieMaxAge" value="100000"/>

</bean>

**Table 18.4. CookieLocaleResolver properties**

| **Property** | **Default** | **Description** |
| --- | --- | --- |
| cookieName | classname + LOCALE | The name of the cookie |
| cookieMaxAge | Integer.MAX\_INT | The maximum time a cookie will stay persistent on the client. If -1 is specified, the cookie will not be persisted; it will only be available until the client shuts down their browser. |
| cookiePath | / | Limits the visibility of the cookie to a certain part of your site. When cookiePath is specified, the cookie will only be visible to that path and the paths below it. |

**18.8.4 SessionLocaleResolver**

The SessionLocaleResolver allows you to retrieve Locale and TimeZone from the session that might be associated with the user’s request. In contrast to CookieLocaleResolver, this strategy stores locally chosen locale settings in the Servlet container’s HttpSession. As a consequence, those settings are just temporary for each session and therefore lost when each session terminates.

Note that there is no direct relationship with external session management mechanisms such as the Spring Session project. This SessionLocaleResolver will simply evaluate and modify corresponding HttpSession attributes against the current HttpServletRequest.

**18.8.5 LocaleChangeInterceptor**

You can enable changing of locales by adding the LocaleChangeInterceptor to one of the handler mappings (see [Section 18.4, “Handler mappings”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-handlermapping)). It will detect a parameter in the request and change the locale. It calls setLocale() on the LocaleResolver that also exists in the context. The following example shows that calls to all \*.view resources containing a parameter named siteLanguage will now change the locale. So, for example, a request for the following URL, <http://www.sf.net/home.view?siteLanguage=nl> will change the site language to Dutch.

<bean id="localeChangeInterceptor"

class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">

<property name="paramName" value="siteLanguage"/>

</bean>

<bean id="localeResolver"

class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="interceptors">

<list>

<ref bean="localeChangeInterceptor"/>

</list>

</property>

<property name="mappings">

<value>/\*\*/\*.view=someController</value>

</property>

</bean>

**18.9 Using themes**

**18.9.1 Overview of themes**

You can apply Spring Web MVC framework themes to set the overall look-and-feel of your application, thereby enhancing user experience. A theme is a collection of static resources, typically style sheets and images, that affect the visual style of the application.

**18.9.2 Defining themes**

To use themes in your web application, you must set up an implementation of the org.springframework.ui.context.ThemeSource interface. The WebApplicationContext interface extends ThemeSource but delegates its responsibilities to a dedicated implementation. By default the delegate will be an org.springframework.ui.context.support.ResourceBundleThemeSource implementation that loads properties files from the root of the classpath. To use a custom ThemeSource implementation or to configure the base name prefix of the ResourceBundleThemeSource, you can register a bean in the application context with the reserved name themeSource. The web application context automatically detects a bean with that name and uses it.

When using the ResourceBundleThemeSource, a theme is defined in a simple properties file. The properties file lists the resources that make up the theme. Here is an example:

styleSheet=/themes/cool/style.css

background=/themes/cool/img/coolBg.jpg

The keys of the properties are the names that refer to the themed elements from view code. For a JSP, you typically do this using the spring:theme custom tag, which is very similar to the spring:message tag. The following JSP fragment uses the theme defined in the previous example to customize the look and feel:

<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<html>

<head>

<link rel="stylesheet" href="<spring:theme code='styleSheet'/>" type="text/css"/>

</head>

<body style="background=<spring:theme code='background'/>">

...

</body>

</html>

By default, the ResourceBundleThemeSource uses an empty base name prefix. As a result, the properties files are loaded from the root of the classpath. Thus you would put the cool.properties theme definition in a directory at the root of the classpath, for example, in /WEB-INF/classes. The ResourceBundleThemeSource uses the standard Java resource bundle loading mechanism, allowing for full internationalization of themes. For example, we could have a /WEB-INF/classes/cool\_nl.properties that references a special background image with Dutch text on it.

**18.9.3 Theme resolvers**

After you define themes, as in the preceding section, you decide which theme to use. The DispatcherServlet will look for a bean named themeResolver to find out which ThemeResolver implementation to use. A theme resolver works in much the same way as a LocaleResolver. It detects the theme to use for a particular request and can also alter the request’s theme. The following theme resolvers are provided by Spring:

**Table 18.5. ThemeResolver implementations**

| **Class** | **Description** |
| --- | --- |
| FixedThemeResolver | Selects a fixed theme, set using the defaultThemeName property. |
| SessionThemeResolver | The theme is maintained in the user’s HTTP session. It only needs to be set once for each session, but is not persisted between sessions. |
| CookieThemeResolver | The selected theme is stored in a cookie on the client. |

Spring also provides a ThemeChangeInterceptor that allows theme changes on every request with a simple request parameter.

**18.10 Spring’s multipart (file upload) support**

**18.10.1 Introduction**

Spring’s built-in multipart support handles file uploads in web applications. You enable this multipart support with pluggable MultipartResolver objects, defined in the org.springframework.web.multipart package. Spring provides one MultipartResolver implementation for use with [*Commons FileUpload*](http://jakarta.apache.org/commons/fileupload) and another for use with Servlet 3.0 multipart request parsing.

By default, Spring does no multipart handling, because some developers want to handle multiparts themselves. You enable Spring multipart handling by adding a multipart resolver to the web application’s context. Each request is inspected to see if it contains a multipart. If no multipart is found, the request continues as expected. If a multipart is found in the request, the MultipartResolver that has been declared in your context is used. After that, the multipart attribute in your request is treated like any other attribute.

**18.10.2 Using a MultipartResolver with *Commons FileUpload***

The following example shows how to use the CommonsMultipartResolver:

<bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

<!-- one of the properties available; the maximum file size in bytes -->

<property name="maxUploadSize" value="100000"/>

</bean>

Of course you also need to put the appropriate jars in your classpath for the multipart resolver to work. In the case of the CommonsMultipartResolver, you need to use commons-fileupload.jar.

When the Spring DispatcherServlet detects a multi-part request, it activates the resolver that has been declared in your context and hands over the request. The resolver then wraps the current HttpServletRequest into a MultipartHttpServletRequest that supports multipart file uploads. Using the MultipartHttpServletRequest, you can get information about the multiparts contained by this request and actually get access to the multipart files themselves in your controllers.

**18.10.3 Using a MultipartResolver with *Servlet 3.0***

In order to use Servlet 3.0 based multipart parsing, you need to mark the DispatcherServlet with a "multipart-config" section in web.xml, or with a javax.servlet.MultipartConfigElement in programmatic Servlet registration, or in case of a custom Servlet class possibly with a javax.servlet.annotation.MultipartConfig annotation on your Servlet class. Configuration settings such as maximum sizes or storage locations need to be applied at that Servlet registration level as Servlet 3.0 does not allow for those settings to be done from the MultipartResolver.

Once Servlet 3.0 multipart parsing has been enabled in one of the above mentioned ways you can add the StandardServletMultipartResolver to your Spring configuration:

<bean id="multipartResolver"

class="org.springframework.web.multipart.support.StandardServletMultipartResolver">

</bean>

**18.10.4 Handling a file upload in a form**

After the MultipartResolver completes its job, the request is processed like any other. First, create a form with a file input that will allow the user to upload a form. The encoding attribute ( enctype="multipart/form-data") lets the browser know how to encode the form as multipart request:

<html>

<head>

<title>Upload a file please</title>

</head>

<body>

<h1>Please upload a file</h1>

<form method="post" action="/form" enctype="multipart/form-data">

<input type="text" name="name"/>

<input type="file" name="file"/>

<input type="submit"/>

</form>

</body>

</html>

The next step is to create a controller that handles the file upload. This controller is very similar to a [normal annotated @Controller](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-controller), except that we use MultipartHttpServletRequest or MultipartFile in the method parameters:

*@Controller*

public class FileUploadController {

*@PostMapping("/form")*

public String handleFormUpload(*@RequestParam("name")* String name,

*@RequestParam("file")* MultipartFile file) {

if (!file.isEmpty()) {

byte[] bytes = file.getBytes();

// store the bytes somewhere

return "redirect:uploadSuccess";

}

return "redirect:uploadFailure";

}

}

Note how the @RequestParam method parameters map to the input elements declared in the form. In this example, nothing is done with the byte[], but in practice you can save it in a database, store it on the file system, and so on.

When using Servlet 3.0 multipart parsing you can also use javax.servlet.http.Part for the method parameter:

*@Controller*

public class FileUploadController {

*@PostMapping("/form")*

public String handleFormUpload(*@RequestParam("name")* String name,

*@RequestParam("file")* Part file) {

InputStream inputStream = file.getInputStream();

// store bytes from uploaded file somewhere

return "redirect:uploadSuccess";

}

}

**18.10.5 Handling a file upload request from programmatic clients**

Multipart requests can also be submitted from non-browser clients in a RESTful service scenario. All of the above examples and configuration apply here as well. However, unlike browsers that typically submit files and simple form fields, a programmatic client can also send more complex data of a specific content type — for example a multipart request with a file and second part with JSON formatted data:

POST /someUrl

Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp

Content-Disposition: form-data; name="meta-data"

Content-Type: application/json; charset=UTF-8

Content-Transfer-Encoding: 8bit

{

"name": "value"

}

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp

Content-Disposition: form-data; name="file-data"; filename="file.properties"

Content-Type: text/xml

Content-Transfer-Encoding: 8bit

... File Data ...

You could access the part named "meta-data" with a @RequestParam("meta-data") String metadata controller method argument. However, you would probably prefer to accept a strongly typed object initialized from the JSON formatted data in the body of the request part, very similar to the way @RequestBody converts the body of a non-multipart request to a target object with the help of an HttpMessageConverter.

You can use the @RequestPart annotation instead of the @RequestParam annotation for this purpose. It allows you to have the content of a specific multipart passed through an HttpMessageConverter taking into consideration the 'Content-Type' header of the multipart:

*@PostMapping("/someUrl")*

public String onSubmit(**@RequestPart("meta-data") MetaData metadata,**

**@RequestPart("file-data") MultipartFile file**) {

// ...

}

Notice how MultipartFile method arguments can be accessed with @RequestParam or with @RequestPart interchangeably. However, the @RequestPart("meta-data") MetaData method argument in this case is read as JSON content based on its 'Content-Type' header and converted with the help of the MappingJackson2HttpMessageConverter.

**18.11 Handling exceptions**

**18.11.1 HandlerExceptionResolver**

Spring HandlerExceptionResolver implementations deal with unexpected exceptions that occur during controller execution. A HandlerExceptionResolver somewhat resembles the exception mappings you can define in the web application descriptor web.xml. However, they provide a more flexible way to do so. For example they provide information about which handler was executing when the exception was thrown. Furthermore, a programmatic way of handling exceptions gives you more options for responding appropriately before the request is forwarded to another URL (the same end result as when you use the Servlet specific exception mappings).

Besides implementing the HandlerExceptionResolver interface, which is only a matter of implementing the resolveException(Exception, Handler) method and returning a ModelAndView, you may also use the provided SimpleMappingExceptionResolver or create @ExceptionHandler methods. The SimpleMappingExceptionResolver enables you to take the class name of any exception that might be thrown and map it to a view name. This is functionally equivalent to the exception mapping feature from the Servlet API, but it is also possible to implement more finely grained mappings of exceptions from different handlers. The @ExceptionHandler annotation on the other hand can be used on methods that should be invoked to handle an exception. Such methods may be defined locally within an @Controller or may apply to many @Controller classes when defined within an @ControllerAdvice class. The following sections explain this in more detail.

**18.11.2 @ExceptionHandler**

The HandlerExceptionResolver interface and the SimpleMappingExceptionResolver implementations allow you to map Exceptions to specific views declaratively along with some optional Java logic before forwarding to those views. However, in some cases, especially when relying on @ResponseBody methods rather than on view resolution, it may be more convenient to directly set the status of the response and optionally write error content to the body of the response.

You can do that with @ExceptionHandler methods. When declared within a controller such methods apply to exceptions raised by @RequestMapping methods of that controller (or any of its sub-classes). You can also declare an @ExceptionHandler method within an @ControllerAdvice class in which case it handles exceptions from @RequestMapping methods from many controllers. Below is an example of a controller-local @ExceptionHandler method:

*@Controller*

public class SimpleController {

// @RequestMapping methods omitted ...

*@ExceptionHandler(IOException.class)*

public ResponseEntity<String> handleIOException(IOException ex) {

// prepare responseEntity

return responseEntity;

}

}

The @ExceptionHandler value can be set to an array of Exception types. If an exception is thrown that matches one of the types in the list, then the method annotated with the matching @ExceptionHandler will be invoked. If the annotation value is not set then the exception types listed as method arguments are used.

Much like standard controller methods annotated with a @RequestMapping annotation, the method arguments and return values of @ExceptionHandler methods can be flexible. For example, the HttpServletRequest can be accessed in Servlet environments and the PortletRequest in Portlet environments. The return type can be a String, which is interpreted as a view name, a ModelAndView object, a ResponseEntity, or you can also add the @ResponseBody to have the method return value converted with message converters and written to the response stream.

**18.11.3 Handling Standard Spring MVC Exceptions**

Spring MVC may raise a number of exceptions while processing a request. The SimpleMappingExceptionResolver can easily map any exception to a default error view as needed. However, when working with clients that interpret responses in an automated way you will want to set specific status code on the response. Depending on the exception raised the status code may indicate a client error (4xx) or a server error (5xx).

The DefaultHandlerExceptionResolver translates Spring MVC exceptions to specific error status codes. It is registered by default with the MVC namespace, the MVC Java config, and also by the DispatcherServlet (i.e. when not using the MVC namespace or Java config). Listed below are some of the exceptions handled by this resolver and the corresponding status codes:

| **Exception** | **HTTP Status Code** |
| --- | --- |
| BindException | 400 (Bad Request) |
| ConversionNotSupportedException | 500 (Internal Server Error) |
| HttpMediaTypeNotAcceptableException | 406 (Not Acceptable) |
| HttpMediaTypeNotSupportedException | 415 (Unsupported Media Type) |
| HttpMessageNotReadableException | 400 (Bad Request) |
| HttpMessageNotWritableException | 500 (Internal Server Error) |
| HttpRequestMethodNotSupportedException | 405 (Method Not Allowed) |
| MethodArgumentNotValidException | 400 (Bad Request) |
| MissingServletRequestParameterException | 400 (Bad Request) |
| MissingServletRequestPartException | 400 (Bad Request) |
| NoHandlerFoundException | 404 (Not Found) |
| NoSuchRequestHandlingMethodException | 404 (Not Found) |
| TypeMismatchException | 400 (Bad Request) |
| MissingPathVariableException | 500 (Internal Server Error) |
| NoHandlerFoundException | 404 (Not Found) |

The DefaultHandlerExceptionResolver works transparently by setting the status of the response. However, it stops short of writing any error content to the body of the response while your application may need to add developer-friendly content to every error response for example when providing a REST API. You can prepare a ModelAndView and render error content through view resolution — i.e. by configuring a ContentNegotiatingViewResolver, MappingJackson2JsonView, and so on. However, you may prefer to use @ExceptionHandler methods instead.

If you prefer to write error content via @ExceptionHandler methods you can extend ResponseEntityExceptionHandler instead. This is a convenient base for @ControllerAdvice classes providing an @ExceptionHandler method to handle standard Spring MVC exceptions and return ResponseEntity. That allows you to customize the response and write error content with message converters. See the ResponseEntityExceptionHandler javadocs for more details.

**18.11.4 Annotating Business Exceptions With @ResponseStatus**

A business exception can be annotated with @ResponseStatus. When the exception is raised, the ResponseStatusExceptionResolver handles it by setting the status of the response accordingly. By default the DispatcherServlet registers the ResponseStatusExceptionResolver and it is available for use.

**18.11.5 Customizing the Default Servlet Container Error Page**

When the status of the response is set to an error status code and the body of the response is empty, Servlet containers commonly render an HTML formatted error page. To customize the default error page of the container, you can declare an <error-page> element in web.xml. Up until Servlet 3, that element had to be mapped to a specific status code or exception type. Starting with Servlet 3 an error page does not need to be mapped, which effectively means the specified location customizes the default Servlet container error page.

<error-page>

<location>/error</location>

</error-page>

Note that the actual location for the error page can be a JSP page or some other URL within the container including one handled through an @Controller method:

When writing error information, the status code and the error message set on the HttpServletResponse can be accessed through request attributes in a controller:

*@Controller*

public class ErrorController {

*@RequestMapping(path = "/error", produces = MediaType.APPLICATION\_JSON\_UTF8\_VALUE)*

*@ResponseBody*

public Map<String, Object> handle(HttpServletRequest request) {

Map<String, Object> map = new HashMap<String, Object>();

map.put("status", request.getAttribute("javax.servlet.error.status\_code"));

map.put("reason", request.getAttribute("javax.servlet.error.message"));

return map;

}

}

or in a JSP:

<%@ page contentType="application/json" pageEncoding="UTF-8"%>

{

status:<%=request.getAttribute("javax.servlet.error.status\_code") %>,

reason:<%=request.getAttribute("javax.servlet.error.message") %>

}

**18.12 Web Security**

The [Spring Security](http://projects.spring.io/spring-security/) project provides features to protect web applications from malicious exploits. Check out the reference documentation in the sections on ["CSRF protection"](http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#csrf), ["Security Response Headers"](http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#headers), and also ["Spring MVC Integration"](http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#mvc). Note that using Spring Security to secure the application is not necessarily required for all features. For example CSRF protection can be added simply by adding the CsrfFilter and CsrfRequestDataValueProcessor to your configuration. See the [Spring MVC Showcase](https://github.com/spring-projects/spring-mvc-showcase/commit/361adc124c05a8187b84f25e8a57550bb7d9f8e4) for an example.

Another option is to use a framework dedicated to Web Security. [HDIV](http://hdiv.org/) is one such framework and integrates with Spring MVC.

**18.13 Convention over configuration support**

For a lot of projects, sticking to established conventions and having reasonable defaults is just what they (the projects) need, and Spring Web MVC now has explicit support for *convention over configuration*. What this means is that if you establish a set of naming conventions and suchlike, you can *substantially* cut down on the amount of configuration that is required to set up handler mappings, view resolvers, ModelAndView instances, etc. This is a great boon with regards to rapid prototyping, and can also lend a degree of (always good-to-have) consistency across a codebase should you choose to move forward with it into production.

Convention-over-configuration support addresses the three core areas of MVC: models, views, and controllers.

**18.13.1 The Controller ControllerClassNameHandlerMapping**

The ControllerClassNameHandlerMapping class is a HandlerMapping implementation that uses a convention to determine the mapping between request URLs and the Controller instances that are to handle those requests.

Consider the following simple Controller implementation. Take special notice of the *name* of the class.

public class **ViewShoppingCartController** implements Controller {

public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {

// the implementation is not hugely important for this example...

}

}

Here is a snippet from the corresponding Spring Web MVC configuration file:

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

<bean id="**viewShoppingCart**" class="x.y.z.ViewShoppingCartController">

<!-- inject dependencies as required... -->

</bean>

The ControllerClassNameHandlerMapping finds all of the various handler (or Controller) beans defined in its application context and strips Controller off the name to define its handler mappings. Thus, ViewShoppingCartController maps to the /viewshoppingcart\* request URL.

Let’s look at some more examples so that the central idea becomes immediately familiar. (Notice all lowercase in the URLs, in contrast to camel-cased Controller class names.)

* WelcomeController maps to the /welcome\* request URL
* HomeController maps to the /home\* request URL
* IndexController maps to the /index\* request URL
* RegisterController maps to the /register\* request URL

In the case of MultiActionController handler classes, the mappings generated are slightly more complex. The Controller names in the following examples are assumed to be MultiActionController implementations:

* AdminController maps to the /admin/\* request URL
* CatalogController maps to the /catalog/\* request URL

If you follow the convention of naming your Controller implementations as xxxController, the ControllerClassNameHandlerMapping saves you the tedium of defining and maintaining a potentially *looooong* SimpleUrlHandlerMapping (or suchlike).

The ControllerClassNameHandlerMapping class extends the AbstractHandlerMapping base class so you can define HandlerInterceptor instances and everything else just as you would with many other HandlerMapping implementations.

**18.13.2 The Model ModelMap (ModelAndView)**

The ModelMap class is essentially a glorified Map that can make adding objects that are to be displayed in (or on) a View adhere to a common naming convention. Consider the following Controller implementation; notice that objects are added to the ModelAndView without any associated name specified.

public class DisplayShoppingCartController implements Controller {

public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {

List cartItems = // get a List of CartItem objects

User user = // get the User doing the shopping

ModelAndView mav = new ModelAndView("displayShoppingCart"); <-- the logical view name

mav.addObject(cartItems); <-- look ma, no name, just the object

mav.addObject(user); <-- and again ma!

return mav;

}

}

The ModelAndView class uses a ModelMap class that is a custom Map implementation that automatically generates a key for an object when an object is added to it. The strategy for determining the name for an added object is, in the case of a scalar object such as User, to use the short class name of the object’s class. The following examples are names that are generated for scalar objects put into a ModelMap instance.

* An x.y.User instance added will have the name user generated.
* An x.y.Registration instance added will have the name registration generated.
* An x.y.Foo instance added will have the name foo generated.
* A java.util.HashMap instance added will have the name hashMap generated. You probably want to be explicit about the name in this case because hashMap is less than intuitive.
* Adding null will result in an IllegalArgumentException being thrown. If the object (or objects) that you are adding could be null, then you will also want to be explicit about the name.

**What, no automatic pluralization?**

Spring Web MVC’s convention-over-configuration support does not support automatic pluralization. That is, you cannot add a List of Person objects to a ModelAndView and have the generated name be people.

This decision was made after some debate, with the "Principle of Least Surprise" winning out in the end.

The strategy for generating a name after adding a Set or a List is to peek into the collection, take the short class name of the first object in the collection, and use that with List appended to the name. The same applies to arrays although with arrays it is not necessary to peek into the array contents. A few examples will make the semantics of name generation for collections clearer:

* An x.y.User[] array with zero or more x.y.User elements added will have the name userList generated.
* An x.y.Foo[] array with zero or more x.y.User elements added will have the name fooList generated.
* A java.util.ArrayList with one or more x.y.User elements added will have the name userList generated.
* A java.util.HashSet with one or more x.y.Foo elements added will have the name fooList generated.
* An *empty* java.util.ArrayList will not be added at all (in effect, the addObject(..) call will essentially be a no-op).

**18.13.3 The View - RequestToViewNameTranslator**

The RequestToViewNameTranslator interface determines a logical View name when no such logical view name is explicitly supplied. It has just one implementation, the DefaultRequestToViewNameTranslator class.

The DefaultRequestToViewNameTranslator maps request URLs to logical view names, as with this example:

public class RegistrationController implements Controller {

public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {

// process the request...

ModelAndView mav = new ModelAndView();

// add data as necessary to the model...

return mav;

// notice that no View or logical view name has been set

}

}

<?xml version="1.0" encoding="UTF-8"?>

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

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.xsd">

<!-- this bean with the well known name generates view names for us -->

<bean id="viewNameTranslator"

class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator"/>

<bean class="x.y.RegistrationController">

<!-- inject dependencies as necessary -->

</bean>

<!-- maps request URLs to Controller names -->

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

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

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

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

</bean>

</beans>

Notice how in the implementation of the handleRequest(..) method no View or logical view name is ever set on the ModelAndView that is returned. The DefaultRequestToViewNameTranslator is tasked with generating a *logical view name* from the URL of the request. In the case of the above RegistrationController, which is used in conjunction with the ControllerClassNameHandlerMapping, a request URL of <http://localhost/registration.html> results in a logical view name of registration being generated by the DefaultRequestToViewNameTranslator. This logical view name is then resolved into the /WEB-INF/jsp/registration.jsp view by the InternalResourceViewResolver bean.

|  |
| --- |
| [Tip] |
| You do not need to define a DefaultRequestToViewNameTranslator bean explicitly. If you like the default settings of the DefaultRequestToViewNameTranslator, you can rely on the Spring Web MVC DispatcherServlet to instantiate an instance of this class if one is not explicitly configured. |

Of course, if you need to change the default settings, then you do need to configure your own DefaultRequestToViewNameTranslator bean explicitly. Consult the comprehensive DefaultRequestToViewNameTranslator javadocs for details on the various properties that can be configured.

**18.14 HTTP caching support**

A good HTTP caching strategy can significantly improve the performance of a web application and the experience of its clients. The 'Cache-Control' HTTP response header is mostly responsible for this, along with conditional headers such as 'Last-Modified' and 'ETag'.

The 'Cache-Control' HTTP response header advises private caches (e.g. browsers) and public caches (e.g. proxies) on how they can cache HTTP responses for further reuse.

An [ETag](http://en.wikipedia.org/wiki/HTTP_ETag) (entity tag) is an HTTP response header returned by an HTTP/1.1 compliant web server used to determine change in content at a given URL. It can be considered to be the more sophisticated successor to the Last-Modified header. When a server returns a representation with an ETag header, the client can use this header in subsequent GETs, in an If-None-Match header. If the content has not changed, the server returns 304: Not Modified.

This section describes the different choices available to configure HTTP caching in a Spring Web MVC application.

**18.14.1 Cache-Control HTTP header**

Spring Web MVC supports many use cases and ways to configure "Cache-Control" headers for an application. While [RFC 7234 Section 5.2.2](https://tools.ietf.org/html/rfc7234#section-5.2.2) completely describes that header and its possible directives, there are several ways to address the most common cases.

Spring Web MVC uses a configuration convention in several of its APIs: setCachePeriod(int seconds):

* A -1 value won’t generate a 'Cache-Control' response header.
* A 0 value will prevent caching using the 'Cache-Control: no-store' directive.
* An n > 0 value will cache the given response for n seconds using the 'Cache-Control: max-age=n' directive.

The [CacheControl](http://docs.spring.io/spring-framework/docs/5.0.0.M1/javadoc-api/org/springframework/http/CacheControl.html) builder class simply describes the available "Cache-Control" directives and makes it easier to build your own HTTP caching strategy. Once built, a CacheControl instance can then be accepted as an argument in several Spring Web MVC APIs.

// Cache for an hour - "Cache-Control: max-age=3600"

CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"

CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,

// public caches should not transform the response

// "Cache-Control: max-age=864000, public, no-transform"

CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS)

.noTransform().cachePublic();

**18.14.2 HTTP caching support for static resources**

Static resources should be served with appropriate 'Cache-Control' and conditional headers for optimal performance. [Configuring a ResourceHttpRequestHandler](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-static-resources) for serving static resources not only natively writes 'Last-Modified' headers by reading a file’s metadata, but also 'Cache-Control' headers if properly configured.

You can set the cachePeriod attribute on a ResourceHttpRequestHandler or use a CacheControl instance, which supports more specific directives:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public-resources/")

.setCacheControl(CacheControl.maxAge(1, TimeUnit.HOURS).cachePublic());

}

}

And in XML:

<mvc:resources mapping="/resources/\*\*" location="/public-resources/">

<mvc:cache-control max-age="3600" cache-public="true"/>

</mvc:resources>

**18.14.3 Support for the Cache-Control, ETag and Last-Modified response headers in Controllers**

Controllers can support 'Cache-Control', 'ETag', and/or 'If-Modified-Since' HTTP requests; this is indeed recommended if a 'Cache-Control' header is to be set on the response. This involves calculating a lastModified long and/or an Etag value for a given request, comparing it against the 'If-Modified-Since' request header value, and potentially returning a response with status code 304 (Not Modified).

As described in [the section called “Using HttpEntity”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-httpentity), controllers can interact with the request/response using HttpEntity types. Controllers returning ResponseEntity can include HTTP caching information in responses like this:

*@GetMapping("/book/{id}")*

public ResponseEntity<Book> showBook(*@PathVariable* Long id) {

Book book = findBook(id);

String version = book.getVersion();

return ResponseEntity

.ok()

.cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))

.eTag(version) // lastModified is also available

.body(book);

}

Doing this will not only include 'ETag' and 'Cache-Control' headers in the response, it will **also convert the response to an HTTP 304 Not Modified response with an empty body** if the conditional headers sent by the client match the caching information set by the Controller.

An @RequestMapping method may also wish to support the same behavior. This can be achieved as follows:

*@RequestMapping*

public String myHandleMethod(WebRequest webRequest, Model model) {

long lastModified = // 1. application-specific calculation

if (request.checkNotModified(lastModified)) {

// 2. shortcut exit - no further processing necessary

return null;

}

// 3. or otherwise further request processing, actually preparing content

model.addAttribute(...);

return "myViewName";

}

There are two key elements here: calling request.checkNotModified(lastModified) and returning null. The former sets the appropriate response status and headers before it returns true. The latter, in combination with the former, causes Spring MVC to do no further processing of the request.

Note that there are 3 variants for this:

* request.checkNotModified(lastModified) compares lastModified with the 'If-Modified-Since' or 'If-Unmodified-Since' request header
* request.checkNotModified(eTag) compares eTag with the 'If-None-Match' request header
* request.checkNotModified(eTag, lastModified) does both, meaning that both conditions should be valid

When receiving conditional 'GET'/'HEAD' requests, checkNotModified will check that the resource has not been modified and if so, it will result in a HTTP 304 Not Modified response. In case of conditional 'POST'/'PUT'/'DELETE' requests, checkNotModified will check that the resource has not been modified and if it has been, it will result in a HTTP 409 Precondition Failed response to prevent concurrent modifications.

**18.14.4 Shallow ETag support**

Support for ETags is provided by the Servlet filter ShallowEtagHeaderFilter. It is a plain Servlet Filter, and thus can be used in combination with any web framework. The ShallowEtagHeaderFilter filter creates so-called shallow ETags (as opposed to deep ETags, more about that later).The filter caches the content of the rendered JSP (or other content), generates an MD5 hash over that, and returns that as an ETag header in the response. The next time a client sends a request for the same resource, it uses that hash as the If-None-Match value. The filter detects this, renders the view again, and compares the two hashes. If they are equal, a 304 is returned.

Note that this strategy saves network bandwidth but not CPU, as the full response must be computed for each request. Other strategies at the controller level (described above) can save network bandwidth and avoid computation.

This filter has a writeWeakETag parameter that configures the filter to write Weak ETags, like this: W/"02a2d595e6ed9a0b24f027f2b63b134d6", as defined in [RFC 7232 Section 2.3](https://tools.ietf.org/html/rfc7232#section-2.3).

You configure the ShallowEtagHeaderFilter in web.xml:

<filter>

<filter-name>etagFilter</filter-name>

<filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>

<!-- Optional parameter that configures the filter to write weak ETags

<init-param>

<param-name>writeWeakETag</param-name>

<param-value>true</param-value>

</init-param>

-->

</filter>

<filter-mapping>

<filter-name>etagFilter</filter-name>

<servlet-name>petclinic</servlet-name>

</filter-mapping>

Or in Servlet 3.0+ environments,

public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

// ...

*@Override*

protected Filter[] getServletFilters() {

return new Filter[] { new ShallowEtagHeaderFilter() };

}

}

See [Section 18.15, “Code-based Servlet container initialization”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-container-config) for more details.

**18.15 Code-based Servlet container initialization**

In a Servlet 3.0+ environment, you have the option of configuring the Servlet container programmatically as an alternative or in combination with a web.xml file. Below is an example of registering a DispatcherServlet:

import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements WebApplicationInitializer {

*@Override*

public void onStartup(ServletContext container) {

XmlWebApplicationContext appContext = new XmlWebApplicationContext();

appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet(appContext));

registration.setLoadOnStartup(1);

registration.addMapping("/");

}

}

WebApplicationInitializer is an interface provided by Spring MVC that ensures your implementation is detected and automatically used to initialize any Servlet 3 container. An abstract base class implementation of WebApplicationInitializer named AbstractDispatcherServletInitializer makes it even easier to register the DispatcherServlet by simply overriding methods to specify the servlet mapping and the location of the DispatcherServlet configuration.

This is recommended for applications that use Java-based Spring configuration:

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

*@Override*

protected Class<?>[] getRootConfigClasses() {

return null;

}

*@Override*

protected Class<?>[] getServletConfigClasses() {

return new Class[] { MyWebConfig.class };

}

*@Override*

protected String[] getServletMappings() {

return new String[] { "/" };

}

}

If using XML-based Spring configuration, you should extend directly from AbstractDispatcherServletInitializer:

public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

*@Override*

protected WebApplicationContext createRootApplicationContext() {

return null;

}

*@Override*

protected WebApplicationContext createServletApplicationContext() {

XmlWebApplicationContext cxt = new XmlWebApplicationContext();

cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

return cxt;

}

*@Override*

protected String[] getServletMappings() {

return new String[] { "/" };

}

}

AbstractDispatcherServletInitializer also provides a convenient way to add Filter instances and have them automatically mapped to the DispatcherServlet:

public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

// ...

*@Override*

protected Filter[] getServletFilters() {

return new Filter[] { new HiddenHttpMethodFilter(), new CharacterEncodingFilter() };

}

}

Each filter is added with a default name based on its concrete type and automatically mapped to the DispatcherServlet.

The isAsyncSupported protected method of AbstractDispatcherServletInitializer provides a single place to enable async support on the DispatcherServlet and all filters mapped to it. By default this flag is set to true.

Finally, if you need to further customize the DispatcherServlet itself, you can override the createDispatcherServlet method.

**18.16 Configuring Spring MVC**

[Section 18.2.1, “Special Bean Types In the WebApplicationContext”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-servlet-special-bean-types) and [Section 18.2.2, “Default DispatcherServlet Configuration”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-servlet-config) explained about Spring MVC’s special beans and the default implementations used by the DispatcherServlet. In this section you’ll learn about two additional ways of configuring Spring MVC. Namely the MVC Java config and the MVC XML namespace.

The MVC Java config and the MVC namespace provide similar default configuration that overrides the DispatcherServlet defaults. The goal is to spare most applications from having to create the same configuration and also to provide higher-level constructs for configuring Spring MVC that serve as a simple starting point and require little or no prior knowledge of the underlying configuration.

You can choose either the MVC Java config or the MVC namespace depending on your preference. Also as you will see further below, with the MVC Java config it is easier to see the underlying configuration as well as to make fine-grained customizations directly to the created Spring MVC beans. But let’s start from the beginning.

**18.16.1 Enabling the MVC Java Config or the MVC XML Namespace**

To enable MVC Java config add the annotation @EnableWebMvc to one of your @Configuration classes:

*@Configuration*

*@EnableWebMvc*

public class WebConfig {

}

To achieve the same in XML use the mvc:annotation-driven element in your DispatcherServlet context (or in your root context if you have no DispatcherServlet context defined):

<?xml version="1.0" encoding="UTF-8"?>

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

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.xsd

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

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

<mvc:annotation-driven/>

</beans>

The above registers a RequestMappingHandlerMapping, a RequestMappingHandlerAdapter, and an ExceptionHandlerExceptionResolver (among others) in support of processing requests with annotated controller methods using annotations such as @RequestMapping, @ExceptionHandler, and others.

It also enables the following:

1. Spring 3 style type conversion through a [ConversionService](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#core-convert) instance in addition to the JavaBeans PropertyEditors used for Data Binding.
2. Support for [formatting](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#format) Number fields using the @NumberFormat annotation through the ConversionService.
3. Support for [formatting](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#format) Date, Calendar, Long, and Joda Time fields using the @DateTimeFormat annotation.
4. Support for [validating](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-validation) @Controller inputs with @Valid, if a JSR-303 Provider is present on the classpath.
5. HttpMessageConverter support for @RequestBody method parameters and @ResponseBody method return values from @RequestMapping or @ExceptionHandler methods.

This is the complete list of HttpMessageConverters set up by mvc:annotation-driven:

* 1. ByteArrayHttpMessageConverter converts byte arrays.
  2. StringHttpMessageConverter converts strings.
  3. ResourceHttpMessageConverter converts to/from org.springframework.core.io.Resource for all media types.
  4. SourceHttpMessageConverter converts to/from a javax.xml.transform.Source.
  5. FormHttpMessageConverter converts form data to/from a MultiValueMap<String, String>.
  6. Jaxb2RootElementHttpMessageConverter converts Java objects to/from XML — added if JAXB2 is present and Jackson 2 XML extension is not present on the classpath.
  7. MappingJackson2HttpMessageConverter converts to/from JSON — added if Jackson 2 is present on the classpath.
  8. MappingJackson2XmlHttpMessageConverter converts to/from XML — added if [Jackson 2 XML extension](https://github.com/FasterXML/jackson-dataformat-xml) is present on the classpath.
  9. AtomFeedHttpMessageConverter converts Atom feeds — added if Rome is present on the classpath.
  10. RssChannelHttpMessageConverter converts RSS feeds — added if Rome is present on the classpath.

See [Section 18.16.12, “Message Converters”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-message-converters) for more information about how to customize these default converters.

|  |
| --- |
| [Note] |
| Jackson JSON and XML converters are created using ObjectMapper instances created by [Jackson2ObjectMapperBuilder](http://docs.spring.io/spring-framework/docs/5.0.0.M1/javadoc-api/org/springframework/http/converter/json/Jackson2ObjectMapperBuilder.html) in order to provide a better default configuration.  This builder customizes Jackson’s default properties with the following ones:   1. [DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES](http://fasterxml.github.io/jackson-databind/javadoc/2.6/com/fasterxml/jackson/databind/DeserializationFeature.html#FAIL_ON_UNKNOWN_PROPERTIES) is disabled. 2. [MapperFeature.DEFAULT\_VIEW\_INCLUSION](http://fasterxml.github.io/jackson-databind/javadoc/2.6/com/fasterxml/jackson/databind/MapperFeature.html#DEFAULT_VIEW_INCLUSION) is disabled.   It also automatically registers the following well-known modules if they are detected on the classpath:   1. [jackson-datatype-jdk7](https://github.com/FasterXML/jackson-datatype-jdk7): support for Java 7 types like java.nio.file.Path. 2. [jackson-datatype-joda](https://github.com/FasterXML/jackson-datatype-joda): support for Joda-Time types. 3. [jackson-datatype-jsr310](https://github.com/FasterXML/jackson-datatype-jsr310): support for Java 8 Date & Time API types. 4. [jackson-datatype-jdk8](https://github.com/FasterXML/jackson-datatype-jdk8): support for other Java 8 types like Optional. |

**18.16.2 Customizing the Provided Configuration**

To customize the default configuration in Java you simply implement the WebMvcConfigurer interface or more likely extend the class WebMvcConfigurerAdapter and override the methods you need:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

// Override configuration methods...

}

To customize the default configuration of <mvc:annotation-driven/> check what attributes and sub-elements it supports. You can view the [Spring MVC XML schema](http://schema.spring.io/mvc/spring-mvc.xsd) or use the code completion feature of your IDE to discover what attributes and sub-elements are available.

**18.16.3 Conversion and Formatting**

By default formatters for Number and Date types are installed, including support for the @NumberFormat and @DateTimeFormat annotations. Full support for the Joda Time formatting library is also installed if Joda Time is present on the classpath. To register custom formatters and converters, override the addFormatters method:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void addFormatters(FormatterRegistry registry) {

// Add formatters and/or converters

}

}

In the MVC namespace the same defaults apply when <mvc:annotation-driven> is added. To register custom formatters and converters simply supply a ConversionService:

<?xml version="1.0" encoding="UTF-8"?>

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

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.xsd

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

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

<mvc:annotation-driven conversion-service="conversionService"/>

<bean id="conversionService"

class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="converters">

<set>

<bean class="org.example.MyConverter"/>

</set>

</property>

<property name="formatters">

<set>

<bean class="org.example.MyFormatter"/>

<bean class="org.example.MyAnnotationFormatterFactory"/>

</set>

</property>

<property name="formatterRegistrars">

<set>

<bean class="org.example.MyFormatterRegistrar"/>

</set>

</property>

</bean>

</beans>

|  |
| --- |
| [Note] |
| See [Section 5.6.4, “FormatterRegistrar SPI”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#format-FormatterRegistrar-SPI) and the FormattingConversionServiceFactoryBean for more information on when to use FormatterRegistrars. |

**18.16.4 Validation**

Spring provides a [Validator interface](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#validator) that can be used for validation in all layers of an application. In Spring MVC you can configure it for use as a global Validator instance, to be used whenever an @Valid or @Validated controller method argument is encountered, and/or as a local Validator within a controller through an @InitBinder method. Global and local validator instances can be combined to provide composite validation.

Spring also [supports JSR-303/JSR-349](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#validation-beanvalidation-overview) Bean Validation via LocalValidatorFactoryBean which adapts the Spring org.springframework.validation.Validator interface to the Bean Validation javax.validation.Validator contract. This class can be plugged into Spring MVC as a global validator as described next.

By default use of @EnableWebMvc or <mvc:annotation-driven> automatically registers Bean Validation support in Spring MVC through the LocalValidatorFactoryBean when a Bean Validation provider such as Hibernate Validator is detected on the classpath.

|  |
| --- |
| [Note] |
| Sometimes it’s convenient to have a LocalValidatorFactoryBean injected into a controller or another class. The easiest way to do that is to declare your own @Bean and also mark it with @Primary in order to avoid a conflict with the one provided with the MVC Java config.  If you prefer to use the one from the MVC Java config, you’ll need to override the mvcValidator method from WebMvcConfigurationSupport and declare the method to explicitly return LocalValidatorFactory rather than Validator. See [Section 18.16.13, “Advanced Customizations with MVC Java Config”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-config-advanced-java) for information on how to switch to extend the provided configuration. |

Alternatively you can configure your own global Validator instance:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public Validator getValidator(); {

// return "global" validator

}

}

and in XML:

<?xml version="1.0" encoding="UTF-8"?>

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

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.xsd

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

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

<mvc:annotation-driven validator="globalValidator"/>

</beans>

To combine global with local validation, simply add one or more local validator(s):

*@Controller*

public class MyController {

*@InitBinder*

protected void initBinder(WebDataBinder binder) {

binder.addValidators(new FooValidator());

}

}

With this minimal configuration any time an @Valid or @Validated method argument is encountered, it will be validated by the configured validators. Any validation violations will automatically be exposed as errors in the BindingResult accessible as a method argument and also renderable in Spring MVC HTML views.

**18.16.5 Interceptors**

You can configure HandlerInterceptors or WebRequestInterceptors to be applied to all incoming requests or restricted to specific URL path patterns.

An example of registering interceptors in Java:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(new LocaleInterceptor());

registry.addInterceptor(new ThemeInterceptor()).addPathPatterns("/\*\*").excludePathPatterns("/admin/\*\*");

registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/secure/\*");

}

}

And in XML use the <mvc:interceptors> element:

<mvc:interceptors>

<bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>

<mvc:interceptor>

<mvc:mapping path="/\*\*"/>

<mvc:exclude-mapping path="/admin/\*\*"/>

<bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path="/secure/\*"/>

<bean class="org.example.SecurityInterceptor"/>

</mvc:interceptor>

</mvc:interceptors>

**18.16.6 Content Negotiation**

You can configure how Spring MVC determines the requested media types from the request. The available options are to check the URL path for a file extension, check the "Accept" header, a specific query parameter, or to fall back on a default content type when nothing is requested. By default the path extension in the request URI is checked first and the "Accept" header is checked second.

The MVC Java config and the MVC namespace register json, xml, rss, atom by default if corresponding dependencies are on the classpath. Additional path extension-to-media type mappings may also be registered explicitly and that also has the effect of whitelisting them as safe extensions for the purpose of RFD attack detection (see [the section called “Suffix Pattern Matching and RFD”](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-ann-requestmapping-rfd) for more detail).

Below is an example of customizing content negotiation options through the MVC Java config:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

configurer.mediaType("json", MediaType.APPLICATION\_JSON);

}

}

In the MVC namespace, the <mvc:annotation-driven> element has a content-negotiation-manager attribute, which expects a ContentNegotiationManager that in turn can be created with a ContentNegotiationManagerFactoryBean:

<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>

<bean id="contentNegotiationManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="mediaTypes">

<value>

json=application/json

xml=application/xml

</value>

</property>

</bean>

If not using the MVC Java config or the MVC namespace, you’ll need to create an instance of ContentNegotiationManager and use it to configure RequestMappingHandlerMapping for request mapping purposes, and RequestMappingHandlerAdapter and ExceptionHandlerExceptionResolver for content negotiation purposes.

Note that ContentNegotiatingViewResolver now can also be configured with a ContentNegotiationManager, so you can use one shared instance throughout Spring MVC.

In more advanced cases, it may be useful to configure multiple ContentNegotiationManager instances that in turn may contain custom ContentNegotiationStrategy implementations. For example you could configure ExceptionHandlerExceptionResolver with a ContentNegotiationManager that always resolves the requested media type to "application/json". Or you may want to plug a custom strategy that has some logic to select a default content type (e.g. either XML or JSON) if no content types were requested.

**18.16.7 View Controllers**

This is a shortcut for defining a ParameterizableViewController that immediately forwards to a view when invoked. Use it in static cases when there is no Java controller logic to execute before the view generates the response.

An example of forwarding a request for "/" to a view called "home" in Java:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/").setViewName("home");

}

}

And the same in XML use the <mvc:view-controller> element:

<mvc:view-controller path="/" view-name="home"/>

**18.16.8 View Resolvers**

The MVC config simplifies the registration of view resolvers.

The following is a Java config example that configures content negotiation view resolution using FreeMarker HTML templates and Jackson as a default View for JSON rendering:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.enableContentNegotiation(new MappingJackson2JsonView());

registry.jsp();

}

}

And the same in XML:

<mvc:view-resolvers>

<mvc:content-negotiation>

<mvc:default-views>

<bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

</mvc:default-views>

</mvc:content-negotiation>

<mvc:jsp/>

</mvc:view-resolvers>

Note however that FreeMarker, Tiles, Groovy Markup and script templates also require configuration of the underlying view technology.

The MVC namespace provides dedicated elements. For example with FreeMarker:

<mvc:view-resolvers>

<mvc:content-negotiation>

<mvc:default-views>

<bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

</mvc:default-views>

</mvc:content-negotiation>

<mvc:freemarker cache="false"/>

</mvc:view-resolvers>

<mvc:freemarker-configurer>

<mvc:template-loader-path location="/freemarker"/>

</mvc:freemarker-configurer>

In Java config simply add the respective "Configurer" bean:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.enableContentNegotiation(new MappingJackson2JsonView());

registry.freeMarker().cache(false);

}

*@Bean*

public FreeMarkerConfigurer freeMarkerConfigurer() {

FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();

configurer.setTemplateLoaderPath("/WEB-INF/");

return configurer;

}

}

**18.16.9 Serving of Resources**

This option allows static resource requests following a particular URL pattern to be served by a ResourceHttpRequestHandler from any of a list of Resource locations. This provides a convenient way to serve static resources from locations other than the web application root, including locations on the classpath. The cache-period property may be used to set far future expiration headers (1 year is the recommendation of optimization tools such as Page Speed and YSlow) so that they will be more efficiently utilized by the client. The handler also properly evaluates the Last-Modified header (if present) so that a 304 status code will be returned as appropriate, avoiding unnecessary overhead for resources that are already cached by the client. For example, to serve resource requests with a URL pattern of /resources/\*\* from a public-resources directory within the web application root you would use:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*").addResourceLocations("/public-resources/");

}

}

And the same in XML:

<mvc:resources mapping="/resources/\*\*" location="/public-resources/"/>

To serve these resources with a 1-year future expiration to ensure maximum use of the browser cache and a reduction in HTTP requests made by the browser:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*").addResourceLocations("/public-resources/").setCachePeriod(31556926);

}

}

And in XML:

<mvc:resources mapping="/resources/\*\*" location="/public-resources/" cache-period="31556926"/>

For more details, see [HTTP caching support for static resources](http://docs.spring.io/spring/docs/5.0.0.M1/spring-framework-reference/htmlsingle/#mvc-caching-static-resources).

The mapping attribute must be an Ant pattern that can be used by SimpleUrlHandlerMapping, and the location attribute must specify one or more valid resource directory locations. Multiple resource locations may be specified using a comma-separated list of values. The locations specified will be checked in the specified order for the presence of the resource for any given request. For example, to enable the serving of resources from both the web application root and from a known path of /META-INF/public-web-resources/ in any jar on the classpath use:

*@EnableWebMvc*

*@Configuration*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/", "classpath:/META-INF/public-web-resources/");

}

}

And in XML:

<mvc:resources mapping="/resources/\*\*" location="/, classpath:/META-INF/public-web-resources/"/>

When serving resources that may change when a new version of the application is deployed it is recommended that you incorporate a version string into the mapping pattern used to request the resources so that you may force clients to request the newly deployed version of your application’s resources. Support for versioned URLs is built into the framework and can be enabled by configuring a resource chain on the resource handler. The chain consists of one more ResourceResolver instances followed by one or more ResourceTransformer instances. Together they can provide arbitrary resolution and transformation of resources.

The built-in VersionResourceResolver can be configured with different strategies. For example a FixedVersionStrategy can use a property, a date, or other as the version. A ContentVersionStrategy uses an MD5 hash computed from the content of the resource (known as "fingerprinting" URLs). Note that the VersionResourceResolver will automatically use the resolved version strings as HTTP ETag header values when serving resources.

ContentVersionStrategy is a good default choice to use except in cases where it cannot be used (e.g. with JavaScript module loaders). You can configure different version strategies against different patterns as shown below. Keep in mind also that computing content-based versions is expensive and therefore resource chain caching should be enabled in production.

Java config example;

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public-resources/")

.resourceChain(true).addResolver(

new VersionResourceResolver().addContentVersionStrategy("/\*\*"));

}

}

XML example:

<mvc:resources mapping="/resources/\*\*" location="/public-resources/">

<mvc:resource-chain>

<mvc:resource-cache/>

<mvc:resolvers>

<mvc:version-resolver>

<mvc:content-version-strategy patterns="/\*\*"/>

</mvc:version-resolver>

</mvc:resolvers>

</mvc:resource-chain>

</mvc:resources>

In order for the above to work the application must also render URLs with versions. The easiest way to do that is to configure the ResourceUrlEncodingFilter which wraps the response and overrides its encodeURL method. This will work in JSPs, FreeMarker, and any other view technology that calls the response encodeURL method. Alternatively, an application can also inject and use directly the ResourceUrlProvider bean, which is automatically declared with the MVC Java config and the MVC namespace.

Webjars are also supported with WebJarsResourceResolver, which is automatically registered when the "org.webjars:webjars-locator" library is on classpath. This resolver allows the resource chain to resolve version agnostic libraries from HTTP GET requests "GET /jquery/jquery.min.js" will return resource "/jquery/1.2.0/jquery.min.js". It also works by rewriting resource URLs in templates <script src="/jquery/jquery.min.js"/> → <script src="/jquery/1.2.0/jquery.min.js"/>.

**18.16.10 Falling Back On the "Default" Servlet To Serve Resources**

This allows for mapping the DispatcherServlet to "/" (thus overriding the mapping of the container’s default Servlet), while still allowing static resource requests to be handled by the container’s default Servlet. It configures a DefaultServletHttpRequestHandler with a URL mapping of "/\*\*" and the lowest priority relative to other URL mappings.

This handler will forward all requests to the default Servlet. Therefore it is important that it remains last in the order of all other URL HandlerMappings. That will be the case if you use <mvc:annotation-driven> or alternatively if you are setting up your own customized HandlerMapping instance be sure to set its order property to a value lower than that of the DefaultServletHttpRequestHandler, which is Integer.MAX\_VALUE.

To enable the feature using the default setup use:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

}

Or in XML:

<mvc:default-servlet-handler/>

The caveat to overriding the "/" Servlet mapping is that the RequestDispatcher for the default Servlet must be retrieved by name rather than by path. The DefaultServletHttpRequestHandler will attempt to auto-detect the default Servlet for the container at startup time, using a list of known names for most of the major Servlet containers (including Tomcat, Jetty, GlassFish, JBoss, Resin, WebLogic, and WebSphere). If the default Servlet has been custom configured with a different name, or if a different Servlet container is being used where the default Servlet name is unknown, then the default Servlet’s name must be explicitly provided as in the following example:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable("myCustomDefaultServlet");

}

}

Or in XML:

<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>

**18.16.11 Path Matching**

This allows customizing various settings related to URL mapping and path matching. For details on the individual options check out the [PathMatchConfigurer](http://docs.spring.io/spring-framework/docs/5.0.0.M1/javadoc-api/org/springframework/web/servlet/config/annotation/PathMatchConfigurer.html) API.

Below is an example in Java config:

*@Configuration*

*@EnableWebMvc*

public class WebConfig extends WebMvcConfigurerAdapter {

*@Override*

public void configurePathMatch(PathMatchConfigurer configurer) {

configurer

.setUseSuffixPatternMatch(true)

.setUseTrailingSlashMatch(false)

.setUseRegisteredSuffixPatternMatch(true)

.setPathMatcher(antPathMatcher())

.setUrlPathHelper(urlPathHelper());

}

*@Bean*

public UrlPathHelper urlPathHelper() {

//...

}

*@Bean*

public PathMatcher antPathMatcher() {

//...

}

}

And the same in XML, use the <mvc:path-matching> element:

<mvc:annotation-driven>

<mvc:path-matching

suffix-pattern="true"

trailing-slash="false"

registered-suffixes-only="true"

path-helper="pathHelper"

path-matcher="pathMatcher"/>

</mvc:annotation-driven>

<bean id="pathHelper" class="org.example.app.MyPathHelper"/>

<bean id="pathMatcher" class="org.example.app.MyPathMatcher"/>

**18.16.12 Message Converters**

Customization of HttpMessageConverter can be achieved in Java config by overriding [configureMessageConverters()](http://docs.spring.io/spring-framework/docs/5.0.0.M1/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurerAdapter.html#configureMessageConverters-java.util.List-) if you want to replace the default converters created by Spring MVC, or by overriding [extendMessageConverters()](http://docs.spring.io/spring-framework/docs/5.0.0.M1/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurerAdapter.html#extendMessageConverters-java.util.List-) if you just want to customize them or add additional converters to the default ones.

Below is an example that adds Jackson JSON and XML converters with a customized ObjectMapper instead of default ones:

*@Configuration*

*@EnableWebMvc*

public class WebConfiguration extends WebMvcConfigurerAdapter {

*@Override*

public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {

Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()

.indentOutput(true)

.dateFormat(new SimpleDateFormat("yyyy-MM-dd"))

.modulesToInstall(new ParameterNamesModule());

converters.add(new MappingJackson2HttpMessageConverter(builder.build()));

converters.add(new MappingJackson2XmlHttpMessageConverter(builder.xml().build()));

}

}

In this example, Jackson2ObjectMapperBuilder is used to create a common configuration for both MappingJackson2HttpMessageConverter and MappingJackson2XmlHttpMessageConverter with indentation enabled, a customized date format and the registration of [jackson-module-parameter-names](https://github.com/FasterXML/jackson-module-parameter-names) that adds support for accessing parameter names (feature added in Java 8).

|  |
| --- |
| [Note] |
| Enabling indentation with Jackson XML support requires [woodstox-core-asl](http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.codehaus.woodstox%22%20AND%20a%3A%22woodstox-core-asl%22) dependency in addition to [jackson-dataformat-xml](http://search.maven.org/#search%7Cga%7C1%7Ca%3A%22jackson-dataformat-xml%22) one. |

Other interesting Jackson modules are available:

1. [jackson-datatype-money](https://github.com/zalando/jackson-datatype-money): support for javax.money types (unofficial module)
2. [jackson-datatype-hibernate](https://github.com/FasterXML/jackson-datatype-hibernate): support for Hibernate specific types and properties (including lazy-loading aspects)

It is also possible to do the same in XML:

<mvc:annotation-driven>

<mvc:message-converters>

<bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">

<property name="objectMapper" ref="objectMapper"/>

</bean>

<bean class="org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter">

<property name="objectMapper" ref="xmlMapper"/>

</bean>

</mvc:message-converters>

</mvc:annotation-driven>

<bean id="objectMapper" class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean"

p:indentOutput="true"

p:simpleDateFormat="yyyy-MM-dd"

p:modulesToInstall="com.fasterxml.jackson.module.paramnames.ParameterNamesModule"/>

<bean id="xmlMapper" parent="objectMapper" p:createXmlMapper="true"/>

**18.16.13 Advanced Customizations with MVC Java Config**

As you can see from the above examples, MVC Java config and the MVC namespace provide higher level constructs that do not require deep knowledge of the underlying beans created for you. Instead it helps you to focus on your application needs. However, at some point you may need more fine-grained control or you may simply wish to understand the underlying configuration.

The first step towards more fine-grained control is to see the underlying beans created for you. In MVC Java config you can see the javadocs and the @Bean methods in WebMvcConfigurationSupport. The configuration in this class is automatically imported through the @EnableWebMvc annotation. In fact if you open @EnableWebMvc you can see the @Import statement.

The next step towards more fine-grained control is to customize a property on one of the beans created in WebMvcConfigurationSupport or perhaps to provide your own instance. This requires two things — remove the @EnableWebMvc annotation in order to prevent the import and then extend from DelegatingWebMvcConfiguration, a subclass of WebMvcConfigurationSupport. Here is an example:

*@Configuration*

public class WebConfig extends DelegatingWebMvcConfiguration {

*@Override*

public void addInterceptors(InterceptorRegistry registry){

// ...

}

*@Override*

*@Bean*

public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {

// Create or let "super" create the adapter

// Then customize one of its properties

}

}

|  |
| --- |
| [Note] |
| An application should have only one configuration extending DelegatingWebMvcConfiguration or a single @EnableWebMvc annotated class, since they both register the same underlying beans.  Modifying beans in this way does not prevent you from using any of the higher-level constructs shown earlier in this section. WebMvcConfigurerAdapter subclasses and WebMvcConfigurer implementations are still being used. |

**18.16.14 Advanced Customizations with the MVC Namespace**

Fine-grained control over the configuration created for you is a bit harder with the MVC namespace.

If you do need to do that, rather than replicating the configuration it provides, consider configuring a BeanPostProcessor that detects the bean you want to customize by type and then modifying its properties as necessary. For example:

*@Component*

public class MyPostProcessor implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean, String name) throws BeansException {

if (bean instanceof RequestMappingHandlerAdapter) {

// Modify properties of the adapter

}

}

}

Note that MyPostProcessor needs to be included in an <component scan/> in order for it to be detected or if you prefer you can declare it explicitly with an XML bean declaration.