17. Web MVC framework

Part V. The Web

§ 17. Web MVC framework

17.1 Introduction to Spring Web MVC framework

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

> **“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
>
> 1. Bob Martin, The Open-Closed Principle (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 9.6.1, “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 need not 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, Velocity 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, a Velocity template model.

17.1.1 Features of Spring Web MVC

Spring Web Flow

Spring Web Flow (SWF) aims to be the best solution for the management of web application page flow.

SWF integrates with existing frameworks like Spring MVC, Struts, and JSF, in both servlet and portlet environments. 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.

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

- *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 and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity 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 Appendix G, *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 Appendix H, *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 5.5.4, “Request, session, and global session scopes”

17.1.2 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. A large body of knowledge and experience exist for the Struts framework. If you can abide Struts' architectural flaws, it can be a viable choice for the web layer; the same applies to WebWork and other web MVC frameworks.

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 a Struts or WebWork action. 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.

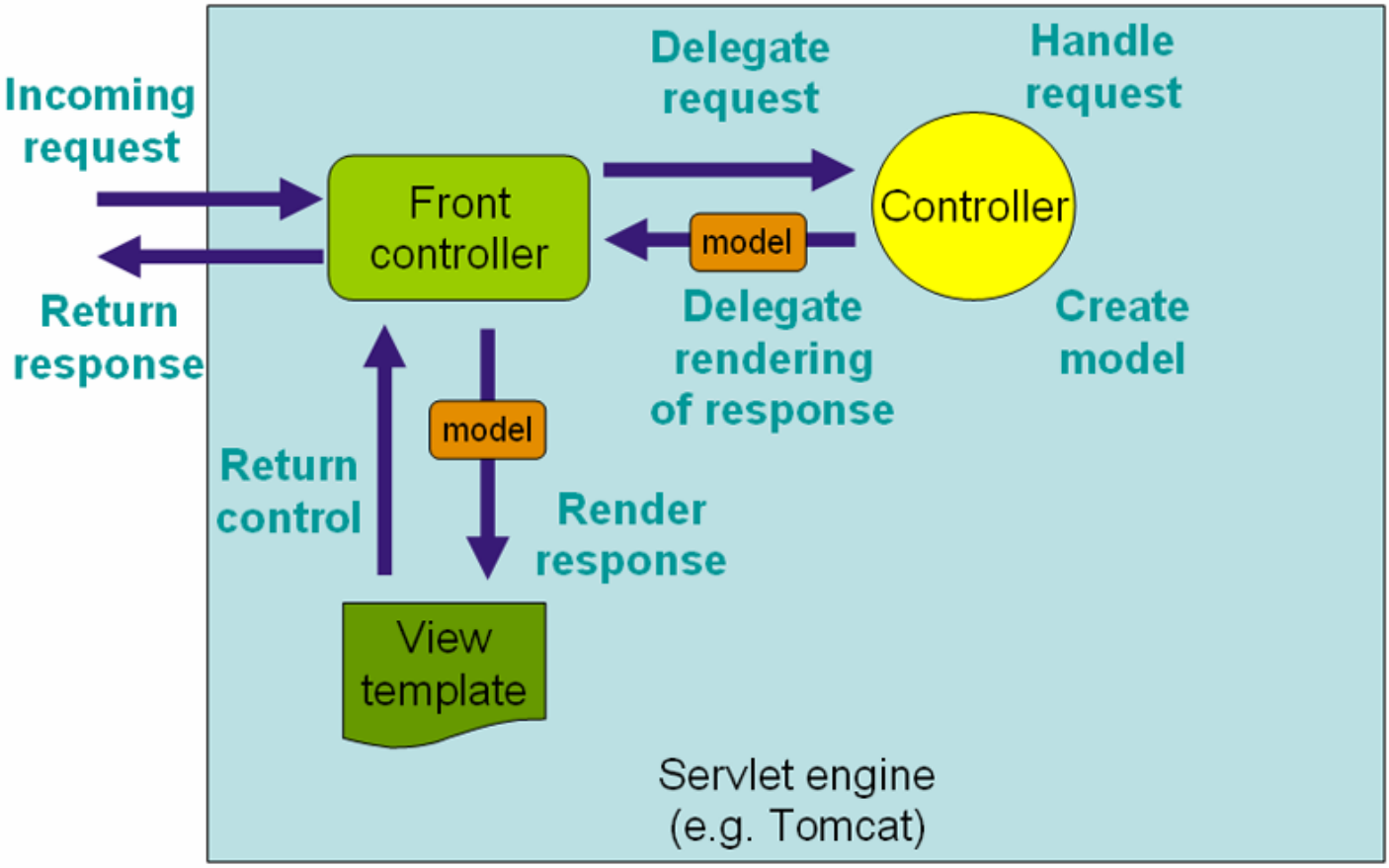
Your registered beans and Spring's services can be at your fingertips even without Spring's Web MVC. Spring does not compete with Struts or WebWork in this scenario. It simply addresses the many areas that the pure web MVC frameworks do not, from bean configuration to data access and transaction handling. 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.

17.2 The `DispatcherServlet`

Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Spring's `DispatcherServlet` however, does more than just that. It is completely integrated with

the Spring IoC container and as such allows you to use every other feature that Spring has.

The request processing workflow of the Spring Web MVC `DispatcherServlet` is illustrated in the following diagram. 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).



The request processing workflow in Spring Web MVC (high level)

The `DispatcherServlet` is an actual `Servlet` (it inherits from the `HttpServlet` base class), and as such is declared in the `web.xml` of your web application. You need to map requests that you want the `DispatcherServlet` to handle, by using a URL mapping in the same `web.xml` file. This is standard Java EE Servlet configuration; the following example shows such a `DispatcherServlet` declaration and mapping:

```
<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>
```

In the preceding example, all requests starting with `/example` will be handled by the `DispatcherServlet` instance named `example`. In a Servlet 3.0+ environment, you also have the option of configuring the Servlet container programmatically. Below is the code based equivalent of the above `web.xml` example:

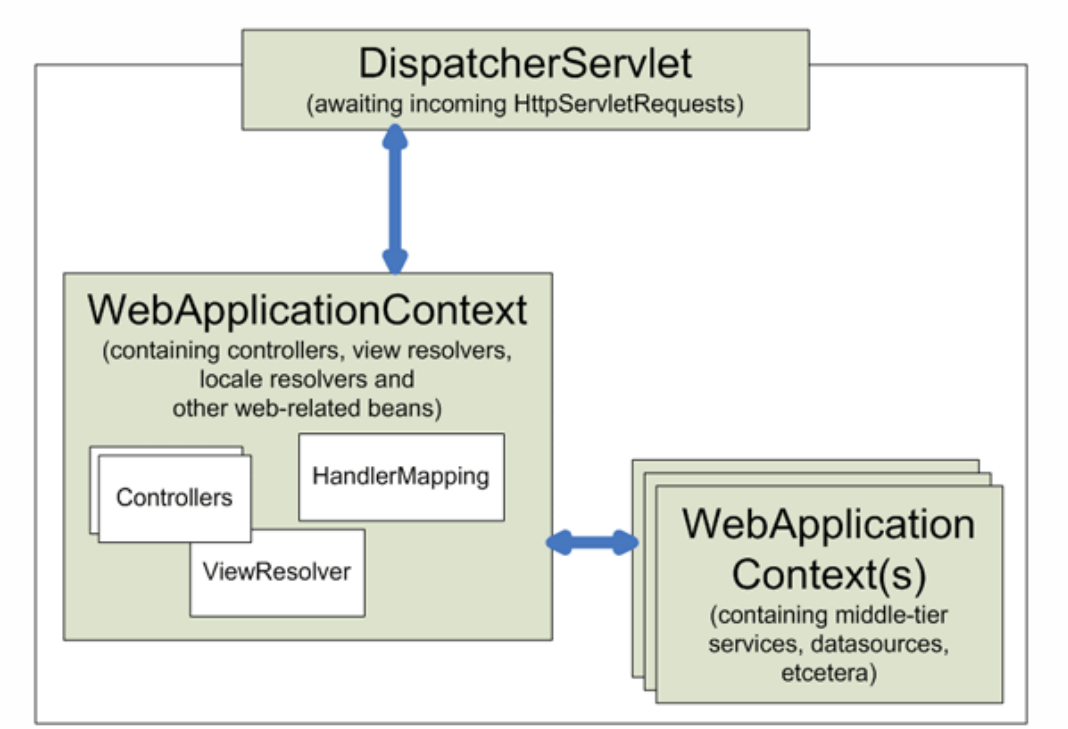
```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new Di
        registration.setLoadOnStartup(1);
        registration.addMapping("/example/*");
    }

}
```

`WebApplicationInitializer` is an interface provided by Spring MVC that ensures your code-based configuration is detected and automatically used to initialize any Servlet 3 container. An abstract base class implementation of this interace named `AbstractDispatcherServletInitializer` makes it even easier to register the `DispatcherServlet` by simply specifying its servlet mapping. See Code-based Servlet container initialization for more details.

The above is only the first step in setting up Spring Web MVC. You now need to configure the various beans used by the Spring Web MVC framework (over and above the `DispatcherServlet` itself).

As detailed in Section 5.14, “Additional Capabilities of the `ApplicationContext`”, `ApplicationContext` instances in Spring can be scoped. In the Web MVC framework, each `DispatcherServlet` has its own `WebApplicationContext`, which inherits all the beans already defined in the root `WebApplicationContext`. These inherited beans can be

overridden in the servlet-specific scope, and you can define new scope-specific beans local to a given Servlet instance.



Context hierarchy in Spring Web MVC

Upon initialization of a `DispatcherServlet`, Spring MVC looks for a file named `[servlet-name]-servlet.xml` in the `WEB-INF` directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.

Consider the following `DispatcherServlet` Servlet configuration (in the `web.xml` file):

```
<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>
```

With the above Servlet configuration in place, you will need to have a file called `/WEB-INF/golfing-servlet.xml` in your application; this file will contain all of your Spring Web MVC-specific components (beans). You can change the exact location of this configuration file through a Servlet initialization parameter (see below for details).

The `WebApplicationContext` is an extension of the plain `ApplicationContext` that has some extra features necessary for web applications. It differs from a normal `ApplicationContext` in that it is capable of resolving themes (see Section 17.9, “Using themes”), 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.

17.2.1 Special Bean Types In the `WebApplicationContext`

The Spring `DispatcherServlet` uses special beans to process requests and render the appropriate views. These beans are part of Spring MVC. You can choose which special beans to use by simply configuring one or more of them in the `WebApplicationContext`. However, you don't need to do that initially since Spring MVC maintains a list of default beans to use if you don't configure any. More on that in the next section. First see the table below listing the special bean types the `DispatcherServlet` relies on.

Table 17.1. Special bean types in the `WebApplicationContext`

Bean type	Explanation
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.

Bean type	Explanation
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	Maps exceptions to views also allowing for more complex exception handling code.
ViewResolver	Resolves logical String-based view names to actual `View` types.
LocaleResolver	Resolves the locale a client is using, in order to be able to offer internationalized views
ThemeResolver	Resolves themes your web application can use, for example, to offer personalized layouts
MultipartResolver	Parses multi-part requests for example to support processing file uploads from HTML forms.
FlashMapManager	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.

17.2.2 Default DispatcherServlet Configuration

As mentioned in the previous section for each special bean the `DispatcherServlet` maintains a list of implementations to use by default. This information is kept in the file `DispatcherServlet.properties` in the package `org.springframework.web.servlet`.

All special beans have some reasonable defaults of their own. Sooner or later though you'll need to customize one or more of the properties these beans provide. For

example it's quite common to configure an `InternalResourceViewResolver` settings its `prefix` property to the parent location of view files.

Regardless of the details, the important concept to understand here is that once you configure a special bean such as an `InternalResourceViewResolver` in your `WebApplicationContext`, you effectively override the list of default implementations that would have been used otherwise for that special bean type. For example if you configure an `InternalResourceViewResolver`, the default list of `ViewResolver` implementations is ignored.

In Section 17.15, “Configuring Spring MVC” 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 should be of help to you.

17.2.3 DispatcherServlet Processing Sequence

After you set up a `DispatcherServlet`, and a request comes in for that specific `DispatcherServlet`, the `DispatcherServlet` starts processing the request as follows:

1. The `WebApplicationContext` is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the key `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE`.
2. The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.
3. The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.
4. If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a `MultipartHttpServletRequest` for further processing by other elements in the process. See Section 17.10, “Spring's multipart (file upload) support” for further information about multipart handling.
5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering.

6. If a model is returned, the view is rendered. If no model is returned, (may be due to a preprocessor or postprocessor intercepting the request, perhaps for security reasons), no view is rendered, because the request could already have been fulfilled.

Handler exception resolvers that are declared in the `WebApplicationContext` pick up exceptions that are thrown during processing of the request. Using these exception resolvers allows you to define custom behaviors to address exceptions.

The Spring `DispatcherServlet` also supports the return of the *last-modification-date*, as specified by the Servlet API. The process of determining the last modification date for a specific request is straightforward: the `DispatcherServlet` looks up an appropriate handler mapping and tests whether the handler that is found implements the `LastModified` interface. If so, the value of the `long getLastModified(request)` method of the `LastModified` interface is returned to the client.

You can customize individual `DispatcherServlet` instances by adding Servlet initialization parameters (`init-param` elements) to the Servlet declaration in the `web.xml` file. See the following table for the list of supported parameters.

Table 17.2. `DispatcherServlet` initialization parameters

Parameter	Explanation
`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.

Parameter	Explanation
`namespace`	Namespace of the `WebApplicationContext`. Defaults to `[servlet-name]-servlet`.

17.3 Implementing Controllers

Controllers provide access to the application behavior that you typically define through a service interface. Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as `@RequestMapping`, `@RequestParam`, `@ModelAttribute`, and so on. This annotation support is available for both Servlet MVC and Portlet MVC. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces. Furthermore, they do not usually have direct dependencies on Servlet or Portlet APIs, although you can easily configure access to Servlet or Portlet facilities.

Available in the samples repository, 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";
    }
}
```

As you can see, the `@Controller` and `@RequestMapping` annotations allow flexible method names and signatures. In this particular example the method accepts a `Model`

and returns a view name as a `String`, but various other method parameters and return values can be used as explained later in this section. `@Controller` and `@RequestMapping` and a number of other annotations form the basis for the Spring MVC implementation. This section documents these annotations and how they are most commonly used in a Servlet environment.

17.3.1 Defining a controller with `@Controller`

The `@Controller` annotation indicates that a particular class serves the role of a *controller*. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to.

The `@Controller` annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects `@RequestMapping` annotations (see the next section).

You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the `@Controller` stereotype also allows for autodetection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them.

To enable autodetection of such annotated controllers, you add component scanning to your configuration. Use the *spring-context* schema as shown in the following XML snippet:

```
<?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>
```

17.3.2 Mapping Requests With `@RequestMapping`

You use the `@RequestMapping` annotation to map URLs such as `/appointments` onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

The following example from the *Petcare* sample shows a controller in a Spring MVC application that uses this annotation:

```
@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(value="/{day}", method = RequestMethod.GET)
    public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Da
        return appointmentBook.getAppointmentsForDay(day);
    }

    @RequestMapping(value="/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";
    }
}
```

In the example, the `@RequestMapping` is used in a number of places. The first usage is on the type (class) level, which indicates that all handling methods on this controller are relative to the `/appointments` path. The `get()` method has a further `@RequestMapping` refinement: it only accepts GET requests, meaning that an HTTP GET for `/appointments` invokes this method. The `post()` has a similar refinement, and the `getNewForm()` combines the definition of HTTP method and path into one, so that GET requests for `appointments/new` are handled by that method.

The `getForDay()` method shows another usage of `@RequestMapping`: URI templates. (See the next section).

A `@RequestMapping` on the class level is not required. Without it, all paths are simply absolute, and not relative. The following example from the *PetClinic* sample application shows a multi-action controller using `@RequestMapping`:

```
@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() {
        return new ModelMap(this.clinic.getVets());
    }

}
```

A common pitfall when working with annotated controller classes happens when applying functionality that requires creating a proxy for the controller object (e.g. `@Transactional` methods). Usually you will introduce an interface for the controller in order to use JDK dynamic proxies. To make this work you must move the `@RequestMapping` annotations, as well as any other type

> and method-level annotations (e.g. `@ModelAttribute`, `@InitBinder`) to the interface as well as the mapping mechanism can only "see" the interface exposed by the proxy. Alternatively, you could activate `proxy-target-class="true"` in the configuration for the functionality applied to the controller (in our transaction scenario in `<tx:annotation-driven />`). Doing so indicates that CGLIB-based subclass proxies should be used instead of interface-based JDK proxies. For more information on various proxying mechanisms see Section 9.6, “Proxying mechanisms”.
> Note however that method argument annotations, e.g. `@RequestParam`, must be present in the method signatures of the controller class.

New Support Classes for `@RequestMapping` methods in Spring MVC 3.1

Spring 3.1 introduced a new set of support classes for `@RequestMapping` methods called `RequestMappingHandlerMapping` and `RequestMappingHandlerAdapter` respectively. They are recommended for use and even required to take advantage of new features in Spring MVC 3.1 and going forward. The new support classes are enabled by default by the MVC namespace and the MVC Java config but must be configured explicitly if using neither. This section describes a few important differences between the old and the new support classes.

Prior to Spring 3.1, type and method-level request mappings were examined in two separate stages -- a controller was selected first by the `DefaultAnnotationHandlerMapping` and the actual method to invoke was narrowed down second by the `AnnotationMethodHandlerAdapter`.

With the new support classes in Spring 3.1, the `RequestMappingHandlerMapping` is the only place where a decision is made about which method should process the request. Think of controller methods as a collection of unique endpoints with mappings for each method derived from type and method-level `@RequestMapping` information.

This enables some new possibilities. For once a `HandlerInterceptor` or a `HandlerExceptionResolver` can now expect the Object-based handler to be a `HandlerMethod`, which allows them to examine the exact method, its parameters and associated annotations. The processing for a URL no longer needs to be split across different controllers.

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 Template Patterns

URI templates can be used for convenient access to selected parts of a URL in a `@RequestMapping` method.

A URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI. The proposed RFC for URI Templates defines how a URI is parameterized. For example, the URI Template `http://www.example.com/users/{userId}` contains the variable *userId*. Assigning the value *fred* to the variable yields `http://www.example.com/users/fred`.

In Spring MVC you can use the `@PathVariable` annotation on a method argument to bind it to the value of a URI template variable:

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
  Owner owner = ownerService.findOwner(ownerId);
  model.addAttribute("owner", owner);
  return "displayOwner";
}
```

The URI Template "`/owners/{ownerId}`" specifies the variable name `ownerId`. When the controller handles this request, the value of `ownerId` is set to the value found in the

appropriate part of the URI. For example, when a request comes in for `/owners/fred`, the value of `ownerId` is `fred`.

To process the @PathVariable annotation, Spring MVC needs to find the matching URI template variable by name. You can specify it in the annotation:

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable("ownerId") String theOwner, Model model) {
  // implementation omitted
}
```

Or if the URI template variable name matches the method argument name you can omit that detail. As long as your code is not compiled without debugging information, Spring MVC will match the method argument name to the URI template variable name:

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
  // implementation omitted
}
```

A method can have any number of `@PathVariable` annotations:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)
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";
}
```

When a `@PathVariable` annotation is used on a `Map<String, String>` argument, the map is populated with all URI template variables.

A URI template can be assembled from type and path level *@RequestMapping* annotations. As a result the `findPet()` method can be invoked with a URL such as `/owners/42/pets/21`.

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

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

```
  public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model)
    // implementation omitted
  }
}
```

A `@PathVariable` argument can be of **any simple type** such as int, long, Date, etc. Spring automatically converts to the appropriate type or throws a `TypeMismatchException` if it fails to do so. You can also register support for parsing additional data types. See the section called “Method Parameters And Type Conversion” and the section called “Customizing `WebDataBinder` initialization”.

URI Template Patterns with Regular Expressions

Sometimes you need more precision in defining URI template variables. Consider the URL `"/spring-web/spring-web-3.0.5.jar"`. How do you break it down into multiple parts?

The `@RequestMapping` annotation supports the use of regular expressions in URI template variables. The syntax is `{varName:regex}` where the first part defines the variable name and the second - the regular expression.For example:

```
@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{extension:\\.[
  public void handle(@PathVariable String version, @PathVariable String extension) {
    // ...
  }
}
```

Path Patterns

In addition to URI templates, the `@RequestMapping` annotation also supports Ant-style path patterns (for example, `/myPath/*.do`). A combination of URI templates and Ant-style globs is also supported (for example, `/owners/*/pets/{petId}`).

Patterns with Placeholders

Patterns in `@RequestMapping` annotations support ${...} placeholders against local properties and/or system properties and environment variables. This may be useful in cases where the path a controller is mapped to may need to be customized through configuration. For more information on placeholders see the Javadoc for `PropertyPlaceholderConfigurer`.

Matrix Variables

The URI specification RFC 3986 defines the possibility of including name-value pairs within path segments. There is no specific term used in the spec. The general "URI path parameters" could be applied although the more unique "Matrix URIs", originating from an old post by Tim Berners-Lee, is also frequently used and fairly well known. Within Spring MVC these are referred to as matrix variables.

Matrix variables can appear in any path segment, each matrix variable separated with a ";" (semicolon). For example: `"/cars;color=red;year=2012"`. Multiple values may be either "," (comma) separated `"color=red,green,blue"` or the variable name may be repeated `"color=red;color=green;color=blue"`.

If a URL is expected to contain matrix variables, the request mapping pattern must represent them with a URI template. This ensures the request can be matched correctly regardless of whether matrix variables are present or not and in what order they are provided.

Below is an example of extracting the matrix variable "q":

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

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

  // petId == 42
  // q == 11

}
```

Since all path segments may contain matrix variables, in some cases you need to be more specific to identify where the variable is expected to be:

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

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)
public void findPet(
    @MatrixVariable(value="q", pathVar="ownerId") int q1,
    @MatrixVariable(value="q", pathVar="petId") int q2) {

  // q1 == 11
  // q2 == 22

}
```

A matrix variable may be defined as optional and a default value specified:

```
// GET /pets/42

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)
  public void findPet(@MatrixVariable(required=true, defaultValue="1") int q) {

    // q == 1

  }
```

All matrix variables may be obtained in a Map:

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

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)
  public void findPet(
        @MatrixVariable Map<String, String> matrixVars,
        @MatrixVariable(pathVar="petId"") Map<String, String> petMatrixVars) {

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

  }
```

Note that to enable the use of matrix variables, you must set the `removeSemicolonContent` property of `RequestMappingHandlerMapping` to `false`. By default it is set to `false`.

In the MVC namespace, the `mvc:annotation-driven` element has an `enableMatrixVariables` attribute that should be set to `true`. By default it is set to `false`.

Consumable Media Types

You can narrow the primary mapping by specifying a list of consumable media types. The request will be matched only if the *Content-Type* request header matches the specified media type. For example:

```
@Controller
@RequestMapping(value = "/pets", method = RequestMethod.POST, consumes="application/json")
public void addPet(@RequestBody Pet pet, Model model) {
    // implementation omitted
}
```

Consumable media type expressions can also be negated as in *!text/plain* to match to all requests other than those with *Content-Type* of *text/plain*.

The *consumes* condition is supported on the type and on the method level. Unlike most other conditions, when used at the type level, method-level consumable types override rather than extend type-level consumable types.

Producible Media Types

You can narrow the primary mapping by specifying a list of producible media types. The request will be matched only if the *Accept* request header matches one of these values. Furthermore, use of the *produces* condition ensures the actual content type used to generate the response respects the media types specified in the *produces* condition. For example:

```
@Controller
@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, produces="application/j
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model) {
    // implementation omitted
}
```

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*.

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.

Request Parameters and Header Values

You can narrow request matching through request parameter conditions such as `"myParam"`, `"!myParam"`, or `"myParam=myValue"`. The first two test for request parameter presence/absence and the third for a specific parameter value. Here is an example with a request parameter value condition:

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

  @RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, params="myParam=myVal
  public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model)
    // implementation omitted
  }
}
```

The same can be done to test for request header presence/absence or to match based on a specific request header value:

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

@RequestMapping(value = "/pets", method = RequestMethod.GET, headers="myHeader=myValue")
  public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model)
    // implementation omitted
  }
}
```

Although you can match to *Content-Type* and *Accept* header values using media type wild cards (for example *"content-type=text/*"* will match to *"text/plain"* and *"text/html"*), it is recommended to use the *consumes* and *produces* conditions respectively instead. They are intended specifically for that purpose.

17.3.3 Defining `@RequestMapping` handler methods

An `@RequestMapping` handler method can have a very flexible signatures. The supported method arguments and return values are described in the following section. Most arguments can be used in arbitrary order with the only exception of `BindingResult` arguments. This is described in the next section.

Spring 3.1 introduced a new set of support classes for `@RequestMapping` methods called `RequestMappingHandlerMapping` and `RequestMappingHandlerAdapter` respectively. They are recommended for use

> and even required to take advantage of new features in Spring MVC 3.1 and going forward. The new support classes are enabled by default from the MVC namespace and with use of the MVC Java config but must be configured explicitly if using neither.

Supported method argument types

The following are the supported method arguments:

- Request or response objects (Servlet API). Choose any specific request or response type, for example `ServletRequest` or `HttpServletRequest`.
- Session object (Servlet API): of type `HttpSession`. An argument of this type enforces the presence of a corresponding session. As a consequence, such an argument is never `null`.

> Session access may not be thread-safe, in particular in a Servlet environment. Consider setting the `RequestMappingHandlerAdapter`'s "synchronizeOnSession" flag to "true" if multiple requests are allowed to access a session concurrently.

- `org.springframework.web.context.request.WebRequest` or `org.springframework.web.context.request.NativeWebRequest`. Allows for generic request parameter access as well as request/session attribute access, without ties to the native Servlet/Portlet API.
- `java.util.Locale` for the current request locale, determined by the most specific locale resolver available, in effect, the configured `LocaleResolver` in a Servlet environment.
- `java.io.InputStream` / `java.io.Reader` for access to the request's content. This value is the raw InputStream/Reader as exposed by the Servlet API.
- `java.io.OutputStream` / `java.io.Writer` for generating the response's content. This value is the raw OutputStream/Writer as exposed by the Servlet API.
- `java.security.Principal` containing the currently authenticated user.
- `@PathVariable` annotated parameters for access to URI template variables. See the section called “URI Template Patterns”.

- `@MatrixVariable` annotated parameters for access to name-value pairs located in URI path segments. See the section called “Matrix Variables”.
- `@RequestParam` annotated parameters for access to specific Servlet request parameters. Parameter values are converted to the declared method argument type. See the section called “Binding request parameters to method parameters with `@RequestParam`”.
- `@RequestHeader` annotated parameters for access to specific Servlet request HTTP headers. Parameter values are converted to the declared method argument type.
- `@RequestBody` annotated parameters for access to the HTTP request body. Parameter values are converted to the declared method argument type using `HttpMessageConverter`s. See the section called “Mapping the request body with the @RequestBody annotation”.
- `@RequestPart` annotated parameters for access to the content of a "multipart/form-data" request part. See Section 17.10.5, “Handling a file upload request from programmatic clients” and Section 17.10, “Spring's multipart (file upload) support”.
- `HttpEntity<?>` parameters for access to the Servlet request HTTP headers and contents. The request stream will be converted to the entity body using `HttpMessageConverter`s. See the section called “Using `HttpEntity<?>`”.
- `java.util.Map` / `org.springframework.ui.Model` / `org.springframework.ui.ModelMap` for enriching the implicit model that is exposed to the web view.
- `org.springframework.web.servlet.mvc.support.RedirectAttributes` to specify the exact set of attributes to use in case of a redirect and also to add flash attributes (attributes stored temporarily on the server-side to make them available to the request after the redirect). `RedirectAttributes` is used instead of the implicit model if the method returns a "redirect:" prefixed view name or `RedirectView`.
- Command or form objects to bind request parameters to bean properties (via setters) or directly to fields, with customizable type conversion, depending on `@InitBinder` methods and/or the HandlerAdapter configuration. See the `webBindingInitializer` property on `RequestMappingHandlerAdapter`. Such command objects along with their validation results will be exposed as model attributes by default, using the command class class name - e.g. model attribute "orderAddress" for a command object of type "some.package.OrderAddress". The `ModelAttribute` annotation can be used on a method argument to customize the model attribute name used.

- `org.springframework.validation.Errors` / `org.springframework.validation.BindingResult` validation results for a preceding command or form object (the immediately preceding method argument).
- `org.springframework.web.bind.support.SessionStatus` status handle for marking form processing as complete, which triggers the cleanup of session attributes that have been indicated by the `@SessionAttributes` annotation at the handler type level.
- `org.springframework.web.util.UriComponentsBuilder` a builder for preparing a URL relative to the current request's host, port, scheme, context path, and the literal part of the servlet mapping.

The `Errors` or `BindingResult` parameters have to follow the model object that is being bound immediately as the method signature might have more that one model object and Spring will create a separate `BindingResult` instance for each of them so the following sample won't work:

Example 17.1. Invalid ordering of BindingResult and @ModelAttribute

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet,
    Model model, BindingResult result) { … }
```

Note, that there is a `Model` parameter in between `Pet` and `BindingResult`. To get this working you have to reorder the parameters as follows:

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet,
    BindingResult result, Model model) { … }
```

Supported method return types

The following are the supported return types:

- A `ModelAndView` object, with the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `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 `HttpMessageConverter`s. See the section called “Mapping the response body with the `@ResponseBody` annotation”.
- A `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 `HttpMessageConverter`s. See the section called “Using `HttpEntity<?>`”.
- 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.
- 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.

Binding request parameters to method parameters with `@RequestParam`

Use the `@RequestParam` annotation to bind request parameters to a method parameter in your controller.

The following code snippet shows the usage:

```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...
```

Parameters using this annotation are required by default, but you can specify that a parameter is optional by setting `@RequestParam`'s `required` attribute to `false` (e.g., `@RequestParam(value="id", required=false)`).

Type conversion is applied automatically if the target method parameter type is not `String`. See the section called “Method Parameters And Type Conversion”.

Mapping the request body with the @RequestBody annotation

The `@RequestBody` method parameter annotation indicates that a method parameter should be bound to the value of the HTTP request body. For example:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws IOException {
  writer.write(body);
}
```

You convert the request body to the method argument by using an `HttpMessageConverter`. `HttpMessageConverter` is responsible for converting from the HTTP request message to an object and converting from an object to the HTTP response body. The `RequestMappingHandlerAdapter` supports the `@RequestBody` annotation with the following default `HttpMessageConverters`:

- `ByteArrayHttpMessageConverter` converts byte arrays.
- `StringHttpMessageConverter` converts strings.
- `FormHttpMessageConverter` converts form data to/from a MultiValueMap<String, String>.
- `SourceHttpMessageConverter` converts to/from a javax.xml.transform.Source.

For more information on these converters, see Message Converters. Also note that if using the MVC namespace or the MVC Java config, a wider range of message converters are registered by default. See Enabling the MVC Java Config or the MVC XML Namespace for more information.

If you intend to read and write XML, you will need to configure the `MarshallingHttpMessageConverter` with a specific `Marshaller` and an `Unmarshaller` implementation from the `org.springframework.oxm` package. The example below shows how to do that directly in your configuration but if your application is configured through the MVC namespace or the MVC Java config see Enabling the MVC Java Config or the MVC XML Namespace instead.

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdap
    <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"/>
```

An `@RequestBody` method parameter can be annotated with `@Valid`, in which case it will be validated using the configured `Validator` instance. When using the MVC namespace or the MVC Java config, a JSR-303 validator is configured automatically assuming a JSR-303 implementation is available on the classpath.

Just like with `@ModelAttribute` parameters, an `Errors` argument can be used to examine the errors. If such an argument is not declared, a `MethodArgumentNotValidException` will be raised. The exception is handled in the `DefaultHandlerExceptionResolver`, which sends a `400` error back to the client.

Also see Enabling the MVC Java Config or the MVC XML Namespace for information on configuring message converters and a validator through the MVC namespace or the MVC Java config.

Mapping the response body with the `@ResponseBody` annotation

The `@ResponseBody` annotation is similar to `@RequestBody`. This annotation can be put on a method and indicates that the return type should be written straight to the HTTP response body (and not placed in a Model, or interpreted as a view name). For example:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
@ResponseBody
public String helloWorld()  {
  return "Hello World";
}
```

The above example will result in the text `Hello World` being written to the HTTP response stream.

As with `@RequestBody`, Spring converts the returned object to a response body by using an `HttpMessageConverter`. For more information on these converters, see the previous section and Message Converters.

Using `HttpEntity<?>`

The `HttpEntity` is similar to `@RequestBody` and `@ResponseBody`. Besides getting access to the request and response body, `HttpEntity` (and the response-specific subclass `ResponseEntity`) also allows access to the request and response headers, like so:

```
@RequestMapping("/something")
public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity) throws UnsupportedEnc
  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);
}
```

The above example gets the value of the `MyRequestHeader` request header, and reads the body as a byte array. It adds the `MyResponseHeader` to the response, writes `Hello World` to the response stream, and sets the response status code to 201 (Created).

As with `@RequestBody` and `@ResponseBody`, Spring uses `HttpMessageConverter` to convert from and to the request and response streams. For more information on these converters, see the previous section and Message Converters.

Using `@ModelAttribute` on a method

The `@ModelAttribute` annotation can be used on methods or on method arguments. This section explains its usage on methods while the next section explains its usage on method arguments.

An `@ModelAttribute` on a method indicates the purpose of that method is to add one or more model attributes. Such methods support the same argument types as `@RequestMapping` methods but cannot be mapped directly to requests. Instead `@ModelAttribute` methods in a controller are invoked before `@RequestMapping` methods, within the same controller. A couple of examples:

```
// Add one attribute
// The return value of the method is added to the model under the name "account"
// You can customize the name via @ModelAttribute("myAccount")

@ModelAttribute
public Account addAccount(@RequestParam String number) {
```

```
    return accountManager.findAccount(number);
}

// Add multiple attributes

@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountManager.findAccount(number));
    // add more ...
}
```

`@ModelAttribute` methods are used to populate the model with commonly needed attributes for example to fill a drop-down with states or with pet types, or to retrieve a command object like Account in order to use it to represent the data on an HTML form. The latter case is further discussed in the next section.

Note the two styles of `@ModelAttribute` methods. In the first, the method adds an attribute implicitly by returning it. In the second, the method accepts a `Model` and adds any number of model attributes to it. You can choose between the two styles depending on your needs.

A controller can have any number of `@ModelAttribute` methods. All such methods are invoked before `@RequestMapping` methods of the same controller.

`@ModelAttribute` methods can also be defined in an `@ControllerAdvice`-annotated class and such methods apply to all controllers. The `@ControllerAdvice` annotation is a component annotation allowing implementation classes to be autodetected through classpath scanning.

What happens when a model attribute name is not explicitly specified? In such cases a default name is assigned to the model attribute based on its type. For example if the method returns an object of type `Account`, the default name used is "account". You can change that through the value of the `@ModelAttribute` annotation. If adding attributes directly to the `Model`, use the appropriate overloaded `addAttribute(..)` method - i.e., with or without an attribute name.

The `@ModelAttribute` annotation can be used on `@RequestMapping` methods as well. In that case the return value of the `@RequestMapping` method is interpreted as a model

attribute rather than as a view name. The view name is derived from view name conventions instead much like for methods returning void — see Section 17.12.3, "The View - `RequestToViewNameTranslator`".

Using `@ModelAttribute` on a method argument

As explained in the previous section `@ModelAttribute` can be used on methods or on method arguments. This section explains its usage on method arguments.

An `@ModelAttribute` on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, the argument's fields should be populated from all request parameters that have matching names. This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute Pet pet) {

}
```

Given the above example where can the Pet instance come from? There are several options:

- It may already be in the model due to use of `@SessionAttributes` — see the section called "Using `@SessionAttributes` to store model attributes in the HTTP session between requests".
- It may already be in the model due to an `@ModelAttribute` method in the same controller — as explained in the previous section.
- It may be retrieved based on a URI template variable and type converter (explained in more detail below).
- It may be instantiated using its default constructor.

An `@ModelAttribute` method is a common way to to retrieve an attribute from the database, which may optionally be stored between requests through the use of `@SessionAttributes`. In some cases it may be convenient to retrieve the attribute by using an URI template variable and a type converter. Here is an example:

```
@RequestMapping(value="/accounts/{account}", method = RequestMethod.PUT)
public String save(@ModelAttribute("account") Account account) {

}
```

In this example the name of the model attribute (i.e. "account") matches the name of a URI template variable. If you register `Converter<String, Account>` that can turn the `String` account value into an `Account` instance, then the above example will work without the need for an `@ModelAttribute` method.

The next step is data binding. The `WebDataBinder` class matches request parameter names — including query string parameters and form fields — to model attribute fields by name. Matching fields are populated after type conversion (from String to the target field type) has been applied where necessary. Data binding and validation are covered in Chapter 7, *Validation, Data Binding, and Type Conversion*. Customizing the data binding process for a controller level is covered in the section called “Customizing `WebDataBinder` initialization”.

As a result of data binding there may be errors such as missing required fields or type conversion errors. To check for such errors add a `BindingResult` argument immediately following the `@ModelAttribute` argument:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

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

    // ...

}
```

With a `BindingResult` you can check if errors were found in which case it's common to render the same form where the errors can be shown with the help of Spring's `<errors>` form tag.

In addition to data binding you can also invoke validation using your own custom validator passing the same `BindingResult` that was used to record data binding errors. That allows for data binding and validation errors to be accumulated in one place and subsequently reported back to the user:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

    new PetValidator().validate(pet, result);
    if (result.hasErrors()) {
        return "petForm";
    }

    // ...
}
```

Or you can have validation invoked automatically by adding the JSR-303 `@Valid` annotation:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {

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

    // ...
}
```

See Section 7.8, “Spring 3 Validation” and Chapter 7, *Validation, Data Binding, and Type Conversion* for details on how to configure and use validation.

Using `@SessionAttributes` to store model attributes in the HTTP session between requests

The type-level `@SessionAttributes` annotation declares session attributes used by a specific handler. 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.

The following code snippet shows the usage of this annotation, specifying the model attribute name:

```
@Controller
@RequestMapping("/editPet.do")
@SessionAttributes("pet")
public class EditPetForm {
    // ...
}
```

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

Specifying redirect and flash attributes

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.

In annotated controllers however the model may contain additional attributes originally added for rendering purposes (e.g. drop-down field values). To gain precise control over the attributes used in a redirect scenario, an `@RequestMapping` method can declare an argument of type `RedirectAttributes` and use it to add attributes for use in `RedirectView`. If the controller method does redirect, the content of `RedirectAttributes` is used. Otherwise the content of the default `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`

The `RedirectAttributes` interface can also be used to add flash attributes. Unlike other redirect attributes, which end up in the target redirect URL, flash attributes are saved in the HTTP session (and hence do not appear in the URL). The model of the controller serving the target redirect URL automatically receives these flash attributes after which they are removed from the session. See Section 17.6, “Using flash attributes” for an overview of the general support for flash attributes in Spring MVC.

Working with `"application/x-www-form-urlencoded"` data

The previous sections covered use of `@ModelAttribute` to support form submission requests from browser clients. The same annotation is recommended for use with requests from non-browser clients as well. However there is one notable difference when it comes to working with HTTP PUT requests. Browsers can submit form data via HTTP GET or HTTP POST. Non-browser clients can also submit forms via HTTP PUT. This presents a challenge because the Servlet specification requires the `ServletRequest.getParameter*()` family of methods to support form field access only for HTTP POST, not for HTTP PUT.

To support HTTP PUT and PATCH requests, the `spring-web` module provides the filter `HttpPutFormContentFilter`, which can be configured in `web.xml`:

```
<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>
```

The above filter intercepts HTTP PUT and PATCH requests with content type `application/x-www-form-urlencoded`, reads the form data from the body of the request, and wraps the `ServletRequest` in order to make the form data available through the `ServletRequest.getParameter*()` family of methods.

As `HttpPutFormContentFilter` consumes the body of the request, it should not be configured for PUT or PATCH URLs that rely on other converters for `application/x-www-form-urlencoded`. This includes `@RequestBody MultiValueMap<String, String>` and `HttpEntity<MultiValueMap<String, String>>`.

Mapping cookie values with the @CookieValue annotation

The `@CookieValue` annotation allows a method parameter to be bound to the value of an HTTP cookie.

Let us consider that the following cookie has been received with an http request:

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

The following code sample demonstrates how to get the value of the `JSESSIONID` cookie:

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@CookieValue("JSESSIONID") String cookie)  {

  //...

}
```

Type conversion is applied automatically if the target method parameter type is not `String`. See the section called “Method Parameters And Type Conversion”.

This annotation is supported for annotated handler methods in Servlet and Portlet environments.

Mapping request header attributes with the @RequestHeader annotation

The `@RequestHeader` annotation allows a method parameter to be bound to a request header.

Here is a sample request header:

```
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
```

The following code sample demonstrates how to get the value of the `Accept-Encoding` and `Keep-Alive` headers:

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,
                              @RequestHeader("Keep-Alive") long keepAlive)  {
```

```
  //...
}
```

Type conversion is applied automatically if the method parameter is not `String`. See the section called “Method Parameters And Type Conversion”.

Built-in support is available for converting a comma-separated string into an array/collection of strings or other types known to the type conversion system. For example a method parameter annotated with `@RequestHeader("Accept")` may be of type `String` but also `String[]` or `List<String>`.

This annotation is supported for annotated handler methods in Servlet and Portlet environments.

Method Parameters And Type Conversion

String-based values extracted from the request including request parameters, path variables, request headers, and cookie values may need to be converted to the target type of the method parameter or field (e.g., binding a request parameter to a field in an `@ModelAttribute` parameter) they're bound to. If the target type is not `String`, Spring automatically converts to the appropriate type. All simple types such as int, long, Date, etc. are supported. You can further customize the conversion process through a `WebDataBinder` (see the section called “Customizing `WebDataBinder` initialization”) or by registering `Formatters` with the `FormattingConversionService` (see Section 7.6, “Spring 3 Field Formatting”).

Customizing `WebDataBinder` initialization

To customize request parameter binding with PropertyEditors through Spring's `WebDataBinder`, you can use `@InitBinder`-annotated methods within your controller, `@InitBinder` methods within an `@ControllerAdvice` class, or provide a custom `WebBindingInitializer`.

Customizing data binding with `@InitBinder`

Annotating controller methods with `@InitBinder` allows you to configure web data binding directly within your controller class. `@InitBinder` identifies methods that initialize the `WebDataBinder` that will be used to populate command and form object arguments of annotated handler methods.

Such init-binder methods support all arguments that `@RequestMapping` supports, except for command/form objects and corresponding validation result objects. Init-binder methods must not have a return value. Thus, they are usually declared as `void`. Typical arguments include `WebDataBinder` in combination with `WebRequest` or `java.util.Locale`, allowing code to register context-specific editors.

The following example demonstrates the use of `@InitBinder` to configure a `CustomDateEditor` for all `java.util.Date` form properties.

```
@Controller
public class MyFormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }


    // ...
}
```

Configuring a custom `WebBindingInitializer`

To externalize data binding initialization, you can provide a custom implementation of the `WebBindingInitializer` interface, which you then enable by supplying a custom bean configuration for an `AnnotationMethodHandlerAdapter`, thus overriding the default configuration.

The following example from the PetClinic application shows a configuration using a custom implementation of the `WebBindingInitializer` interface, `org.springframework.samples.petclinic.web.ClinicBindingInitializer`, which configures PropertyEditors required by several of the PetClinic controllers.

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdap
    <property name="cacheSeconds" value="0" />
```

```
    <property name="webBindingInitializer">
        <bean class="org.springframework.samples.petclinic.web.ClinicBindingInitializer" />
    </property>
</bean>
```

Customizing data binding with externalized `@InitBinder` methods

`@InitBinder` methods can also be defined in an `@ControllerAdvice`-annotated class in which case they apply to all controllers. This provides an alternative to using a `WebBindingInitializer`.

The `@ControllerAdvice` annotation is a component annotation allowing implementation classes to be autodetected through classpath scanning.

Support for the 'Last-Modified' Response Header To Facilitate Content Caching

An `@RequestMapping` method may wish to support `'Last-Modified'` HTTP requests, as defined in the contract for the Servlet API's `getLastModified` method, to facilitate content caching. This involves calculating a lastModified `long` 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). An annotated controller method can achieve that 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 to note: calling `request.checkNotModified(lastModified)` and returning `null`. The former sets the response status to 304 before it returns `true`. The latter, in combination with the former, causes Spring MVC to do no further processing of the request.

17.3.4 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 separate thread. Meanwhile the main Servlet container thread is 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 with the value returned by the `Callable`. Here is an example controller method:

```
@RequestMapping(method=RequestMethod.POST)
public Callable<String> processUpload(final MultipartFile file) {

  return new Callable<String>() {
    public Object call() throws Exception {
      // ...
      return "someView";
    }
  };
}
```

A second option is for the controller to return an instance of `DeferredResult`. In this case the return value will also be produced from a separate thread. However, that thread is not known to Spring MVC. For example the result may be produced in response to some external event such as a JMS message, a scheduled task, etc. Here is an example controller method:

```
@RequestMapping("/quotes")
@ResponseBody
public DeferredResult<String> quotes() {
  DeferredResult<String> deferredResult = new DeferredResult<String>();
  // Save the deferredResult in in-memory queue ...
  return deferredResult;
}

// In some other thread...
deferredResult.setResult(data);
```

This may be difficult to understand without any knowledge of the Servlet 3 async processing feature. It would certainly help to read up on it. At a very minimum consider the following basic facts:

- 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 allowing some other thread to complete processing.
- The call to `request.startAsync()` returns an `AsyncContext`, which can be used for further control over async processing. For example it provides the method `dispatch`, which can be called from an application thread in order to "dispatch" the request back to the Servlet container. An async dispatch is similar to a forward except it is made from one (application) thread to another (Servlet container) thread whereas a forward occurs synchronously in the same (Servlet container) thread.
- `ServletRequest` provides access to the current `DispatcherType`, which can be used to distinguish if a `Servlet` or a `Filter` is processing on the initial request processing thread and when it is processing in an async dispatch.

With the above in mind, the following is the sequence of events for async request processing with a `Callable`: (1) Controller returns a `Callable`, (2) Spring MVC starts async processing and submits the `Callable` to a `TaskExecutor` for processing in a separate thread, (3) the `DispatcherServlet` and all Filter's exit the request processing thread but the response remains open, (4) the `Callable` produces a result and Spring MVC dispatches the request back to the Servlet container, (5) the `DispatcherServlet` is invoked again and processing resumes with the asynchronously produced result from the `Callable`. The exact sequencing of (2), (3), and (4) may vary depending on the speed of execution of the concurrent threads.

The sequence of events for async request processing with a `DeferredResult` is the same in principal except it's up to the application to produce the asynchronous result from some thread: (1) Controller returns a `DeferredResult` and saves it in some in-memory queue or list where it can be accessed, (2) Spring MVC starts async processing, (3) the `DispatcherServlet` and all configured Filter's exit the request processing thread but the response remains open, (4) the application sets the `DeferredResult` from some thread and Spring MVC dispatches the request back to the Servlet container, (5) the `DispatcherServlet` is invoked again and processing resumes with the asynchronously produced result.

Explaining the motivation for async request processing and when or why to use it are beyond the scope of this document. For further information you may wish to read this

blog post series.

Exception Handling for Async Requests

What happens if a `Callable` returned from a controller method raises an Exception while being executed? The effect is similar to what happens when any controller method raises an exception. It is handled by a matching `@ExceptionHandler` method in the same controller or by one of the configured `HandlerExceptionResolver` instances.

Under the covers, when a `Callable` raises an Exception, Spring MVC still dispatches to the Servlet container to resume processing. The only difference is that the result of executing the `Callable` is an `Exception` that must be processed with the configured `HandlerExceptionResolver` instances.

When using a `DeferredResult`, you have a choice of calling its `setErrorResult(Object)` method and provide an `Exception` or any other Object you'd like to use as the result. If the result is an `Exception`, it will be processed with a matching `@ExceptionHandler` method in the same controller or with any configured `HandlerExceptionResolver` instance.

Intercepting Async Requests

An existing `HandlerInterceptor` can implement `AsyncHandlerInterceptor`, which provides one additional method `afterConcurrentHandlingStarted`. It is invoked after async processing starts and when the initial request processing thread is being exited. See the Javadoc of `AsyncHandlerInterceptor` for more details on that.

Further options for async request lifecycle callbacks are provided directly on `DeferredResult`, which has the methods `onTimeout(Runnable)` and `onCompletion(Runnable)`. Those are called when the async request is about to time out or has completed respectively. The timeout event can be handled by setting the `DeferredResult` to some value. The completion callback however is final and the result can no longer be set.

Similar callbacks are also available with a `Callable`. However, you will need to wrap the `Callable` in an instance of `WebAsyncTask` and then use that to register the timeout

and completion callbacks. Just like with `DeferredResult`, the timeout event can be handled and a value can be returned while the completion event is final.

You can also register a `CallableProcessingInterceptor` or a `DeferredResultProcessingInterceptor` globally through the MVC Java config or the MVC namespace. Those interceptors provide a full set of callbacks and apply every time a `Callable` or a `DeferredResult` is used.

Configuration for Async Request Processing

Servlet 3 Async Config

To use Servlet 3 async request processing, you need to update `web.xml` to version 3.0:

```
<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/
    version="3.0">

    ...

  </web-app>
```

The `DispatcherServlet` and any `Filter` configuration need to have the `<async-supported>true</async-supported>` sub-element. Additionally, any `Filter` that also needs to get involved in async dispatches should also be configured to support the ASYNC dispatcher type. Note that it is safe to enable the ASYNC dispatcher type for all filters provided with the Spring Framework since they will not get involved in async dispatches unless needed.

If using Servlet 3, Java based configuration, e.g. 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 `AbstractAnnotationConfigDispatcherServletInitializer`, which automatically set those options and make it very easy to register `Filter` instances.

Spring MVC Async Config

The MVC Java config and the MVC namespace both provide options for configuring async 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`.

17.3.5 Testing Controllers

The `spring-test` module offers first class support for testing annotated controllers. See Section 11.3.6, “Spring MVC Test Framework”.

17.4 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. `HandlerInterceptor`s are discussed in Section 17.4.1, “Intercepting requests with a `HandlerInterceptor`”.

`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.RequestMappingHandlerMa
    <property name="interceptors">
      <bean class="example.MyInterceptor"/>
    </property>
  </bean>

<beans>
```

17.4.1 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.RequestMappingHandler
        <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;
        } else {
            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.

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.

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 Enabling the MVC Java Config or the MVC XML Namespace.

17.5 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, Velocity templates and XSLT views, for example. See Chapter 18, *View technologies* 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.

17.5.1 Resolving views with the `ViewResolver` interface

As discussed in Section 17.3, "Implementing Controllers", 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 17.3. 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`.

`ViewResolver`	Description
`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 Javadocs for the `UrlBasedViewResolver` class for details.
`VelocityViewResolver` / `FreeMarkerViewResolver`	Convenient subclass of `UrlBasedViewResolver` that supports `VelocityView` (in effect, Velocity templates) or `FreeMarkerView` ,respectively, 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 17.5.4, "`ContentNegotiatingViewResolver`".

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.

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 Velocity template is modified), you can use the `removeFromCache(String viewName, Locale loc)` method.

17.5.2 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.InternalResourceViewR
  <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 `VelocityViewResolver` and some others. Check the Javadoc for the 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!

17.5.3 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 Velocity, 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`ed 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` issues an `HttpServletResponse.sendRedirect()` call that returns to the client browser as an HTTP redirect. 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 annotated controller can declare an argument of type `RedirectAttributes` and use it to specify the exact attributes to make available to `RedirectView`. If the controller method decides to redirect, the content of `RedirectAttributes` is used. Otherwise the content of the model is used.

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:

```
@RequestMapping(value = "/files/{path}", method = RequestMethod.POST)
public String upload(...) {
    // ...
    return "redirect:files/{path}";
}
```

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 next section discusses this process.

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.

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.

17.5.4 `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 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.

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 the resolution of a view based on a file extension, use the `ContentNegotiatingViewResolver` bean property `mediaTypes` to specify a mapping of file extensions to media types. For more information on the algorithm used to determine the request media type, refer to the API documentation for `ContentNegotiatingViewResolver`.

Here is an example configuration of a `ContentNegotiatingViewResolver:`

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="atom" value="application/atom+xml"/>
      <entry key="html" value="text/html"/>
      <entry key="json" value="application/json"/>
    </map>
  </property>
  <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.springsource.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" 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.

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>();

    @RequestMapping(value="/content", method=RequestMethod.GET)
    public ModelAndView getContent() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("content");
        mav.addObject("sampleContentList", contentList);
        return mav;
    }

}
```

17.6 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 .

17.7 Building `URI`s

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/{book

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}").b
            .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
            .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()
```

17.8 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 17.4.1, “Intercepting requests with a `HandlerInterceptor`” 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.

17.8.1 `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.

17.8.2 `CookieLocaleResolver`

This locale resolver inspects a `Cookie` that might exist on the client to see if a locale is specified. If so, it uses the specified locale. 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 d
    <property name="cookieMaxAge" value="100000">

</bean>
```

Table 17.4. `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 his or her 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.

17.8.3 `SessionLocaleResolver`

The `SessionLocaleResolver` allows you to retrieve locales from the session that might be associated with the user's request.

17.8.4 `LocaleChangeInterceptor`

You can enable changing of locales by adding the `LocaleChangeInterceptor` to one of the handler mappings (see Section 17.4, “Handler mappings”). 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>
```

17.9 Using themes

17.9.1 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.

17.9.2 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.

17.9.3 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 17.5. `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.

17.10 Spring's multipart (file upload) support

17.10.1 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* 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.

17.10.2 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.

17.10.3 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>
```

17.10.4 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`, except that we use `MultipartHttpServletRequest` or `MultipartFile` in the method parameters:

```
@Controller
public class FileUploadController {

    @RequestMapping(value = "/form", method = RequestMethod.POST)
    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";
       } else {
           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 {

    @RequestMapping(value = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name,
        @RequestParam("file") Part file) {

        InputStream inputStream = file.getInputStream();
        // store bytes from uploaded file somewhere

        return "redirect:uploadSuccess";
    }

}
```

17.10.5 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:

```
@RequestMapping(value="/someUrl", method = RequestMethod.POST)
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`.

17.11 Handling exceptions

17.11.1 `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 globally to all `@RequestMapping` methods when defined within an `@ControllerAdvice` class. The following sections explain this in more detail.

17.11.2 `@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 contoroller (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 any controller. The `@ControllerAdvice` annotation is a component annotation, which can be used with classpath scanning. It is automatically enabled when using the MVC namespace and the MVC Java config, or otherwise depending on whether the `ExceptionHandlerExceptionResolver` is configured or not. 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.

17.11.3 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 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)
`NoSuchRequestHandlingMethodException`	404 (Not Found)
`TypeMismatchException`	400 (Bad Request)

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`, `MappingJacksonJsonView`, 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 Javadoc of `ResponseEntityExceptionHandler` for more details.

17.11.4 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.

17.11.5 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(value="/error", produces="application/json")
    @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") %>
}
```

17.12 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.

17.12.1 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 respon
        // 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 `xxx`**Controller**, 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.

17.12.2 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 respon

        List cartItems = // get a List of CartItem objects
        User user = // get the User doing the shopping

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

        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).

17.12.3 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 respon
        // 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.ControllerClassNameHandlerMappi

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

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 Javadoc for the `DefaultRequestToViewNameTranslator` class for details of the various properties that can be configured.

17.13 ETag support

An 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`.

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. This filter will not save processing power, as the view is still rendered. The only thing it saves is bandwidth, as the rendered response is not sent back over the wire.

You configure the `ShallowEtagHeaderFilter` in `web.xml`:

```
<filter>
  <filter-name>etagFilter</filter-name>
    <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>etagFilter</filter-name>
  <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

17.14 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 Di
        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:

```
public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitialize

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { MyWebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}
```

The above example is for an application that uses Java-based Spring configuration. If using XML-based Spring configuration, 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`.

17.15 Configuring Spring MVC

Section 17.2.1, “Special Bean Types In the `WebApplicationContext`” and Section 17.2.2, “Default DispatcherServlet Configuration” 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 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.

17.15.1 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:

```
<?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 instance in addition to the JavaBeans PropertyEditors used for Data Binding.
2. Support for formatting Number fields using the `@NumberFormat` annotation through the `ConversionService`.
3. Support for formatting Date, Calendar, Long, and Joda Time fields using the `@DateTimeFormat` annotation.
4. Support for validating @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:
   - `ByteArrayHttpMessageConverter` converts byte arrays.
   - `StringHttpMessageConverter` converts strings.
   - `ResourceHttpMessageConverter` converts to/from `org.springframework.core.io.Resource` for all media types.
   - `SourceHttpMessageConverter` converts to/from a `javax.xml.transform.Source`.
   - `FormHttpMessageConverter` converts form data to/from a `MultiValueMap<String, String>`.
   - `Jaxb2RootElementHttpMessageConverter` converts Java objects to/from XML — added if JAXB2 is present on the classpath.
   - `MappingJackson2HttpMessageConverter` (or `MappingJacksonHttpMessageConverter`) converts to/from JSON — added if Jackson 2 (or Jackson) is present on the

classpath.

- `AtomFeedHttpMessageConverter` converts Atom feeds — added if Rome is present on the classpath.
- `RssChannelHttpMessageConverter` converts RSS feeds — added if Rome is present on the classpath.

17.15.2 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. Below is an example of some of the available methods to override. See `WebMvcConifgurer` for a list of all methods and the Javadoc for further details:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  protected void addFormatters(FormatterRegistry registry) {
    // Add formatters and/or converters
  }

  @Override
  public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
    // Configure the list of HttpMessageConverters to use
  }

}
```

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 or use the code completion feature of your IDE to discover what attributes and sub-elements are available. The sample below shows a subset of what is available:

```
<mvc:annotation-driven conversion-service="conversionService">
    <mvc:message-converters>
        <bean class="org.example.MyHttpMessageConverter"/>
        <bean class="org.example.MyOtherHttpMessageConverter"/>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="conversionService" class="org.springframework.format.support.FormattingConversionS
```

```
    <property name="formatters">
        <list>
            <bean class="org.example.MyFormatter"/>
            <bean class="org.example.MyOtherFormatter"/>
        </list>
    </property>
</bean>
```

17.15.3 Configuring 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("/**").excludePathPatter
    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>
        <mapping path="/**"/>
        <exclude-mapping path="/admin/**"/>
        <bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor" />
    </mvc:interceptor>
    <mvc:interceptor>
        <mapping path="/secure/*"/>
        <bean class="org.example.SecurityInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>
```

17.15.4 Configuring Content Negotiation

Staring with Spring Framework 3.2, you can configure how Spring MVC determines the requested media types from the client for request mapping as well as for content negotiation purposes. The available options are to check the file extension in the request URI, the "Accept" header, a request parameter, as well as to fall back on a default content type. By default, file extension in the request URI is checked first and the "Accept" header is checked next.

For file extensions in the request URI, the MVC Java config and the MVC namespace, automatically register extensions such as `.json`, `.xml`, `.rss`, and `.atom` if the corresponding dependencies such as Jackson, JAXB2, or Rome are present on the classpath. Additional extensions may be not need to be registered explicitly if they can be discovered via `ServletContext.getMimeType(String)` or the *Java Activation Framework* (see `javax.activation.MimetypesFileTypeMap`).

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.favorPathExtension(false).favorParameter(true);
  }
}
```

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.ContentNegotiatio
    <property name="favorPathExtension" value="false" />
    <property name="favorParameter" value="true" />
    <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 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.

17.15.5 Configuring 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"/>
```

17.15.6 Configuring 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/").
  }
```

```
}
```

And in XML:

```
<mvc:resources mapping="/resources/**" location="/public-resources/" cache-period="31556926"
```

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. Such a version string can be parameterized and accessed using SpEL so that it may be easily managed in a single place when deploying new versions.

As an example, let's consider an application that uses a performance-optimized custom build (as recommended) of the Dojo JavaScript library in production, and that

the build is generally deployed within the web application at a path of `/public-resources/dojo/dojo.js`. Since different parts of Dojo may be incorporated into the custom build for each new version of the application, the client web browsers need to be forced to re-download that custom-built `dojo.js` resource any time a new version of the application is deployed. A simple way to achieve this would be to manage the version of the application in a properties file, such as:

```
application.version=1.0.0
```

and then to make the properties file's values accessible to SpEL as a bean using the `util:properties` tag:

```
<util:properties id="applicationProps" location="/WEB-INF/spring/application.properties"/>
```

With the application version now accessible via SpEL, we can incorporate this into the use of the `resources` tag:

```
<mvc:resources mapping="/resources-#{applicationProps['application.version']}/**" location="
```

In Java, you can use the `@PropertySouce` annotation and then inject the `Environment` abstraction for access to all defined properties:

```
@Configuration
@EnableWebMvc
@PropertySource("/WEB-INF/spring/application.properties")
public class WebConfig extends WebMvcConfigurerAdapter {

  @Inject Environment env;

  @Override
  public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources-" + env.getProperty("application.version") + "/*
      .addResourceLocations("/public-resources/");
  }

}
```

and finally, to request the resource with the proper URL, we can take advantage of the Spring JSP tags:

```
<spring:eval expression="@applicationProps['application.version']" var="applicationVersion"/
```

```
<spring:url value="/resources-{applicationVersion}" var="resourceUrl">
    <spring:param name="applicationVersion" value="${applicationVersion}"/>
</spring:url>

<script src="${resourceUrl}/dojo/dojo.js" type="text/javascript"> </script>
```

17.15.7 mvc:default-servlet-handler

This tag 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"/>
```

17.15.8 More Spring Web MVC Resources

See the following links and pointers for more resources about Spring Web MVC:

- There are many excellent articles and tutorials that show how to build web applications with Spring MVC. Read them at the Spring Documentation page.
- “Expert Spring Web MVC and Web Flow” by Seth Ladd and others (published by Apress) is an excellent hard copy source of Spring Web MVC goodness.

17.15.9 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 Javadoc 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 directly from `WebMvcConfigurationSupport`. Here is an example:

```
@Configuration
public class WebConfig extends WebMvcConfigurationSupport {

  @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 that modifying beans in this way does not prevent you from using any of the higher-level constructs shown earlier in this section.

17.15.10 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 BeansExcept
```

```
    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.