17. Web MVC framework

17.1 Introduction to Spring Web MVC framework

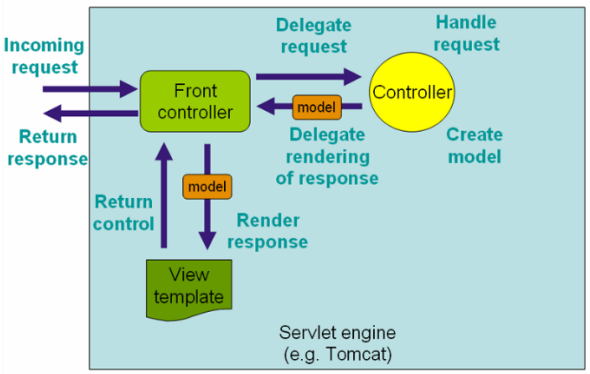
The Spring Web model-view-controller (MVC) framework is designed around a `DispatcherServlet` that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for uploading files. The default handler is based on the `@Controller` 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.

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.

17.2 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:

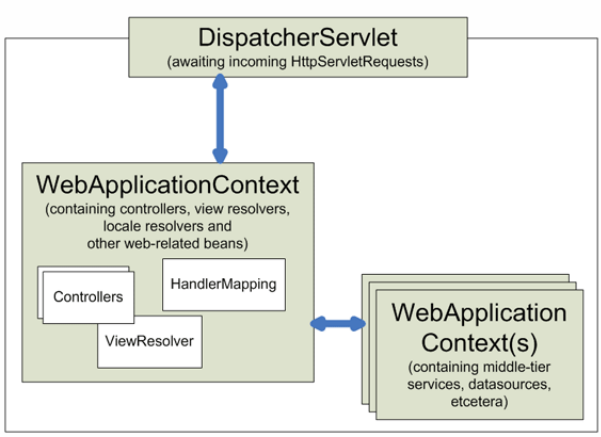
```
1.  <web-app>
2.
3.      <servlet>
4.          <servlet-name>example</servlet-name>
5.          <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
6.          <load-on-startup>1</load-on-startup>
7.      </servlet>
8.
9.      <servlet-mapping>
10.         <servlet-name>example</servlet-name>
11.         <url-pattern>/example/*</url-pattern>
12.     </servlet-mapping>
13.
14. </web-app>
15.
```

In the preceding example, all requests starting with `/example` will be handled by the `DispatcherServlet` instance named `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):

```
1.  <web-app>
2.
3.      <servlet>
4.          <servlet-name>golfing</servlet-name>
5.          <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
6.          <load-on-startup>1</load-on-startup>
7.      </servlet>
8.
9.      <servlet-mapping>
10.         <servlet-name>golfing</servlet-name>
11.         <url-pattern>/golfing/*</url-pattern>
12.     </servlet-mapping>
13.
14. </web-app>
15.
```

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.
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.
`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.

```
1.  @Controller
2.  public class HelloWorldController {
3.
4.      @RequestMapping("/helloWorld")
5.      public String helloWorld(Model model) {
6.          model.addAttribute("message", "Hello World!");
7.          return "helloWorld";
8.      }
9.  }
10.
```

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:

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.      xmlns:p="http://www.springframework.org/schema/p"
5.      xmlns:context="http://www.springframework.org/schema/context"
6.      xsi:schemaLocation="
7.          http://www.springframework.org/schema/beans
8.          http://www.springframework.org/schema/beans/spring-beans.xsd
9.          http://www.springframework.org/schema/context
10.         http://www.springframework.org/schema/context/spring-context.xsd">
11.
12.     <context:component-scan base-package="org.springframework.samples.petclinic.web"/>
13.
14.     <!-- ... -->
15.
16. </beans>
17.
```

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:

```
1.  @Controller
2.  @RequestMapping("/appointments")
3.  public class AppointmentsController {
4.
5.      private final AppointmentBook appointmentBook;
6.
7.      @Autowired
8.      public AppointmentsController(AppointmentBook appointmentBook) {
9.          this.appointmentBook = appointmentBook;
```

```
10.     }
11.
12.     @RequestMapping(method = RequestMethod.GET)
13.     public Map<String, Appointment> get() {
14.         return appointmentBook.getAppointmentsForToday();
15.     }
16.
17.     @RequestMapping(value="/{day}", method = RequestMethod.GET)
18.     public Map<String, Appointment> getForDay(@PathVariable
    @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
19.         return appointmentBook.getAppointmentsForDay(day);
20.     }
21.
22.     @RequestMapping(value="/new", method = RequestMethod.GET)
23.     public AppointmentForm getNewForm() {
24.         return new AppointmentForm();
25.     }
26.
27.     @RequestMapping(method = RequestMethod.POST)
28.     public String add(@Valid AppointmentForm appointment, BindingResult result) {
29.         if (result.hasErrors()) {
30.             return "appointments/new";
31.         }
32.         appointmentBook.addAppointment(appointment);
33.         return "redirect:/appointments";
34.     }
35. }
36.
```

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`:

```
1.  @Controller
2.  public class ClinicController {
3.
4.      private final Clinic clinic;
5.
6.      @Autowired
7.      public ClinicController(Clinic clinic) {
8.          this.clinic = clinic;
9.      }
10.
11.     @RequestMapping("/")
12.     public void welcomeHandler() {
13.     }
14.
15.     @RequestMapping("/vets")
16.     public ModelMap vetsHandler() {
17.         return new ModelMap(this.clinic.getVets());
18.     }}
```

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=myValue")
  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

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:

```
1.  @Controller
2.  @RequestMapping("/pets")
3.  @SessionAttributes("pet")
4.  public class EditPetForm {
5.
6.      // ...
7.
8.      @RequestMapping(method = RequestMethod.GET)
9.      public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
10.         Pet pet = this.clinic.loadPet(petId);
11.         model.addAttribute("pet", pet);
12.         return "petForm";
13.     }
14.
15.     // ...
16.
```

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.RequestMapping
HandlerAdapter">
    <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.MarshallingHttpMessageConvert
er">
  <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 UnsupportedEncodingException {
  String requestHeader =
requestEntity.getHeaders().getFirst("MyRequestHeader"));
  byte[] requestBody = requestEntity.getBody();
  // do something with request header and body

  HttpHeaders responseHeaders = new HttpHeaders();
  responseHeaders.set("MyResponseHeader", "MyValue");
  return new ResponseEntity<String>("Hello World", responseHeaders,
HttpStatus.CREATED);
}
```

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) {

}
```