The Web

Version 5.0.0.BUILD-SNAPSHOT

Table of Contents

This part of the documentation covers support for web applications. As of Spring Framework 5.0 web applications can run on a traditional Servlet stack (Servlet API + Servlet container) or on a reactive stack (Reactive Streams API + non-blocking runtime). The first few chapters cover the Servlet-based Spring MVC web framework including Views, CORS, and WebSocket support. Subsequent chapters cover the Spring WebFlux reactive web framework including its functional programming model.

Chapter 1. Spring Web MVC framework

1.1. Introduction

Spring Web MVC is the Servlet-based, web framework included in the Spring Framework. Its name is based on the name of the module, "spring-webmvc", but most people call it simply Spring MVC.

The Spring Framework also includes the reactive, Spring WebFlux web framework that does not depend on the Servlet API but can run on Servlet containers (via Servlet 3.1 non-blocking I/O) or on other non-blocking runtimes such as Netty or Undertow.

1.2. The DispatcherServlet

Spring MVC, like many other web frameworks, is designed around the front controller pattern with a central `Servlet`, the `DispatcherServlet`, dispatching incoming requests to registered handlers for processing requests, providing convenient mapping and exception handling facilities.

The `DispatcherServlet` provides the shared algorithm for processing requests while actual work is performed by configurable, delegate components. This model is very flexible and it can be used with just about any workflow, with the installation of the appropriate delegate components.

The `DispatcherServlet` uses Spring configuration to discover the delegate components it needs to perform handler mapping, view resolution, and much more (see Special Bean Types). As an actual `Servlet` it also needs to be declared and mapped according to the Servlet specification. This can be done through code-based configuration or in `web.xml`.

Below is an example of code-based configuration. Note that `WebApplicationInitializer` is an interface provided by Spring MVC that ensures it is auto-detected by the Servlet container (see Code-based, Servlet container initialization for more details):

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

  @Override
  public void onStartup(ServletContext servletCxt) {

    // Load Spring web application configuration
    AnnotationConfigWebApplicationContext cxt = new
AnnotationConfigWebApplicationContext();
    cxt.register(AppConfig.class);
    cxt.refresh();

    // Create DispatcherServlet
    DispatcherServlet servlet = new DispatcherServlet(cxt);

    // Register and map the Servlet
    ServletRegistration.Dynamic registration = servletCxt.addServlet("app", servlet);
    registration.setLoadOnStartup(1);
    registration.addMapping("/app/*");
  }

}
```

In addition to using the ServletContext API directly as shown above, you can also extend the convenient base class `AbstractAnnotationConfigDispatcherServletInitializer` and override specific methods to customize it. An example of that is shown in the next section WebApplicationContext Hierarchy.

Below is the `web.xml` equivalent of the above code-based example:

```
<web-app>

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

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/app-context.xml</param-value>
  </context-param>

  <servlet>
    <servlet-name>app</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value></param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>app</servlet-name>
    <url-pattern>/app/*</url-pattern>
  </servlet-mapping>

</web-app>
```

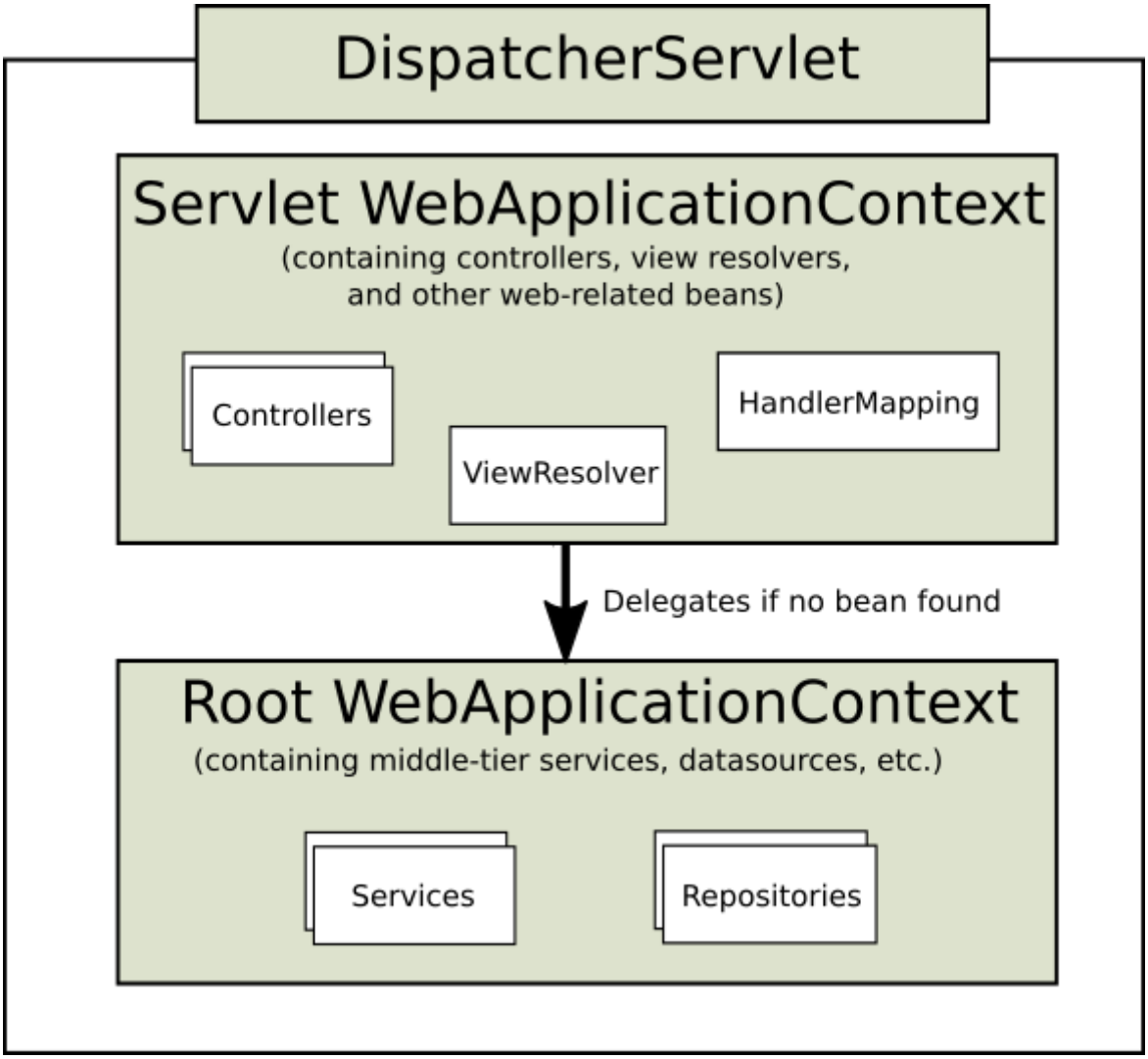
1.2.1. WebApplicationContext Hierarchy

`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 Using themes), and that it knows which Servlet it is associated with (by having a link to the `ServletContext`). `WebApplicationContext` is bound to the `ServletContext` and by using static methods on the `RequestContextUtils` class you can always look up the `WebApplicationContext` if you need access to it.

For many applications, a single `WebApplicationContext` is simple and sufficient. However it is also possible to set up a context hierarchy where one root `WebApplicationContext` is shared across multiple `DispatcherServlet` instances, or other `Servlet`, each with its own `WebApplicationContext` configuration — see Additional Capabilities of the ApplicationContext for more on the context hierarchy feature of Spring.

The root `WebApplicationContext` should contain infrastructure beans, e.g. data repositories or business services, that need to be shared across multiple `Servlet` instances. These beans are effectively inherited and could be overridden, or rather re-declared, in the Servlet-specific

`WebApplicationContext` which for the most part contains beans local to the given `Servlet` as shown in the below diagram:



Below is example configuration of how to set up a `WebApplicationContext` hierarchy:

```
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

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

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class[] { App1Config.class };
  }

  @Override
  protected String[] getServletMappings() {
    return new String[] { "/app1/*" };
  }
}
```

And the `web.xml` equivalent:

```
<web-app>

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

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

  <servlet>
    <servlet-name>app1</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/app1-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>app1</servlet-name>
    <url-pattern>/app1/*</url-pattern>
  </servlet-mapping>

</web-app>
```

1.2.2. Special Bean Types In the WebApplicationContext

The `DispatcherServlet` delegates to special beans to process requests and render the appropriate responses. By "special beans" we mean Spring-managed Object instances that implement specific framework contracts listed in the table further below in this section. Spring MVC provides built-in implementations of these contracts so all you need to do is configure them in your Spring configuration. It is however possible to customize, extend, or completely replace those built-in implementations.

The table below lists special bean types the `DispatcherServlet` depends on and delegates to.

Table 1. Special bean types in the WebApplicationContext

Bean type	Explanation
HandlerMapping	Map a request to a handler along with a list of `HandlerInterceptor`s for pre- and post-processing. The mapping is 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 how the handler is actually invoked. For example, invoking an annotated controller requires resolving various annotations. The main purpose of a `HandlerAdapter` is to shield the `DispatcherServlet` from such details.
HandlerExceptionResolver	Strategy to resolve exceptions possibly mapping them to handlers, or to HTML error views, or other.
ViewResolver	Resolves logical String-based view names returned from a handler to an actual `View` to render to the response with.
LocaleResolver & LocaleContextResolver	Resolves the `Locale` a client is using and possibly their time zone, in order to be able to offer internationalized views
ThemeResolver	Resolves themes your web application can use, for example, to offer personalized layouts
MultipartResolver	Abstraction for parsing a multi-part request (e.g. browser form file upload) with the help of some multipart parsing library.
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.

1.2.3. `DispatcherServlet` Configuration

There are more than one ways to actually configure the `DispatcherServlet` with the special bean types listed in the previous section.

If there are no beans of a given type in the `WebApplicationContext` by default the `DispatcherServlet` will refer to a list of default implementations to use in the file DispatcherServlet.properties.

Applications can explicitly declare the special beans to use to take over the defaults. However for most applications the MVC Java config or the MVC XML namespace are the best starting point. Each creates the necessary Spring configuration for the `DispatcherServlet` and also provide a higher-level API to configure Spring MVC without having to understand all the details. See Configuring Spring MVC for more details.

Spring Boot uses the MVC Java config to configure Spring MVC and provides many extra options and conveniences on top.

1.2.4. DispatcherServlet Processing Sequence

The `DispatcherServlet` processes requests as follows:

- 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`.
- 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.
- 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.
- 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 Multipart (file upload) support for further information about multipart handling.
- 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. Or alternatively for annotated controllers, the response may be rendered (within the `HandlerAdapter`) instead of returning a view.
- 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.

The `HandlerExceptionResolver` beans declared in the `WebApplicationContext` are used to resolve exceptions thrown during request processing. Those exception resolvers allow customizing the logic to address exceptions. See Handling exceptions for more details.

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 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`.

1.3. 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. 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 APIs, although you can easily configure access to Servlet facilities if needed.

Available in the spring-projects Org on Github, 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.

1.3.1. 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>
```

1.3.2. 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 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
<strong>@RequestMapping("/appointments")</strong>
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

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

    <strong>@RequestMapping(method = RequestMethod.GET)</strong>
    public Map<String, Appointment> get() {
        return appointmentBook.getAppointmentsForToday();
    }

    <strong>@RequestMapping(path = "/{day}", method = RequestMethod.GET)</strong>
    public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO
.DATE) Date day, Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    <strong>@RequestMapping(path = "/new", method = RequestMethod.GET)</strong>
    public AppointmentForm getNewForm() {
        return new AppointmentForm();
    }

    <strong>@RequestMapping(method = RequestMethod.POST)</strong>
    public String add(@Valid AppointmentForm appointment, BindingResult result) {
        if (result.hasErrors()) {
            return "appointments/new";
        }
        appointmentBook.addAppointment(appointment);
        return "redirect:/appointments";
    }
}
```

In the above example, `@RequestMapping` is used in a number of places. The first usage is on the type (class) level, which indicates that all handler methods in 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 `add()` 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 URI Template Patterns).

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;
    }

    <strong>@RequestMapping("/")</strong>
    public void welcomeHandler() {
    }

    <strong>@RequestMapping("/vets")</strong>
    public ModelMap vetsHandler() {
        return new ModelMap(this.clinic.getVets());
    }

}
```

The above example does not specify `GET` vs. `PUT`, `POST`, and so forth, because `@RequestMapping` maps all HTTP methods by default. Use `@RequestMapping(method=GET)` or `@GetMapping` to narrow the mapping.

Composed @RequestMapping Variants

Spring Framework 4.3 introduces the following method-level *composed* variants of the `@RequestMapping` annotation that help to simplify mappings for common HTTP methods and better express the semantics of the annotated handler method. For example, a `@GetMapping` can be read as a `GET` `@RequestMapping`.

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

The following example shows a modified version of the `AppointmentsController` from the previous section that has been simplified with *composed* `@RequestMapping` annotations.

```
@Controller
<strong>@RequestMapping("/appointments")</strong>
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

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

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

    <strong>@GetMapping("/{day}")</strong>
    public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO
.DATE) Date day, Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    <strong>@GetMapping("/new")</strong>
    public AppointmentForm getNewForm() {
        return new AppointmentForm();
    }

    <strong>@PostMapping</strong>
    public String add(@Valid AppointmentForm appointment, BindingResult result) {
        if (result.hasErrors()) {
            return "appointments/new";
        }
        appointmentBook.addAppointment(appointment);
        return "redirect:/appointments";
    }
}
```

@Controller and AOP Proxying

In some cases a controller may need to be decorated with an AOP proxy at runtime. One example is if you choose to have `@Transactional` annotations directly on the controller. When this is the case, for controllers specifically, we recommend using class-based proxying. This is typically the default choice with controllers. However if a controller must implement an interface that is not a Spring Context callback (e.g. `InitializingBean`, `*Aware`, etc), you may need to explicitly configure class-based proxying. For example with `<tx:annotation-driven/>`, change to `<tx:annotation-driven proxy-target-class="true"/>`.

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. For example `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:

```
@GetMapping("/owners/{ownerId}")
public String findOwner(<strong>@PathVariable</strong> 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:

```
@GetMapping("/owners/{ownerId}")
public String findOwner(<strong>@PathVariable("ownerId")</strong>
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 compiled with debugging information or the `-parameters` compiler flag on Java 8, Spring MVC will match the method argument name to the URI template variable name:

```
@GetMapping("/owners/{ownerId}")
public String findOwner(<strong>@PathVariable</strong> String ownerId,
Model model) {
    // implementation omitted
}
```

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

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public String findPet(<strong>@PathVariable</strong> String ownerId, <strong
>@PathVariable</strong> 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 method level *@RequestMapping* annotations. As a result the `findPet()` method can be invoked with a URL such as `/owners/42/pets/21`.

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

    @RequestMapping(<strong>"/pets/{petId}"</strong>)
    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 Method Parameters And Type Conversion and 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:\\.[a-z]+}")
public void handle(@PathVariable String version, @PathVariable String extension) {
    // ...
}
```

Path Patterns

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

Path Pattern Comparison

When a URL matches multiple patterns, a sort is used to find the most specific match.

A pattern with a lower count of URI variables and wild cards is considered more specific. For example `/hotels/{hotel}/*` has 1 URI variable and 1 wild card and is considered more specific than `/hotels/{hotel}/**` which as 1 URI variable and 2 wild cards.

If two patterns have the same count, the one that is longer is considered more specific. For example `/foo/bar*` is longer and considered more specific than `/foo/*`.

When two patterns have the same count and length, the pattern with fewer wild cards is considered more specific. For example `/hotels/{hotel}` is more specific than `/hotels/*`.

There are also some additional special rules:

- The **default mapping pattern** `/**` is less specific than any other pattern. For example `/api/{a}/{b}/{c}` is more specific.
- A **prefix pattern** such as `/public/**` is less specific than any other pattern that doesn't contain double wildcards. For example `/public/path3/{a}/{b}/{c}` is more specific.

For the full details see `AntPatternComparator` in `AntPathMatcher`. Note that the PathMatcher can be customized (see Path Matching in the section on configuring Spring MVC).

Path 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 javadocs of the `PropertyPlaceholderConfigurer` class.

Suffix Pattern Matching

By default Spring MVC performs `".*"` suffix pattern matching so that a controller mapped to `/person` is also implicitly mapped to `/person.*`. This makes it easy to request different representations of a resource through the URL path (e.g. `/person.pdf`, `/person.xml`).

Suffix pattern matching can be turned off or restricted to a set of path extensions explicitly registered for content negotiation purposes. This is generally recommended to minimize ambiguity with common request mappings such as `/person/{id}` where a dot might not represent a file extension, e.g. `/person/joe@email.com` vs `/person/joe@email.com.json`. Furthermore as explained in the note below suffix pattern matching as well as content negotiation may be used in some circumstances to attempt malicious attacks and there are good reasons to restrict them meaningfully.

See Path Matching for suffix pattern matching configuration and also Content Negotiation for content negotiation configuration.

Suffix Pattern Matching and RFD

Reflected file download (RFD) attack was first described in a paper by Trustwave in 2014. The attack is similar to XSS in that it relies on input (e.g. query parameter, URI variable) being reflected in the response. However instead of inserting JavaScript into HTML, an RFD attack relies on the browser switching to perform a download and treating the response as an executable script if double-clicked based on the file extension (e.g. .bat, .cmd).

In Spring MVC `@ResponseBody` and `ResponseEntity` methods are at risk because they can render different content types which clients can request including via URL path extensions. Note however

that neither disabling suffix pattern matching nor disabling the use of path extensions for content negotiation purposes alone are effective at preventing RFD attacks.

For comprehensive protection against RFD, prior to rendering the response body Spring MVC adds a `Content-Disposition:inline;filename=f.txt` header to suggest a fixed and safe download file. This is done only if the URL path contains a file extension that is neither whitelisted nor explicitly registered for content negotiation purposes. However it may potentially have side effects when URLs are typed directly into a browser.

Many common path extensions are whitelisted by default. Furthermore REST API calls are typically not meant to be used as URLs directly in browsers. Nevertheless applications that use custom `HttpMessageConverter` implementations can explicitly register file extensions for content negotiation and the Content-Disposition header will not be added for such extensions. See Content Negotiation.

This was originally introduced as part of work for CVE-2015-5211. Below are additional recommendations from the report:

- Encode rather than escape JSON responses. This is also an OWASP XSS recommendation. For an example of how to do that with Spring see spring-jackson-owasp.
- Configure suffix pattern matching to be turned off or restricted to explicitly registered suffixes only.
- Configure content negotiation with the properties "useJaf" and "ignoreUnknownPathExtensions" set to false which would result in a 406 response for URLs with unknown extensions. Note however that this may not be an option if URLs are naturally expected to have a dot towards the end.
- Add `X-Content-Type-Options: nosniff` header to responses. Spring Security 4 does this by default.

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

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

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable(name="q", pathVar="ownerId") int q1,
        @MatrixVariable(name="q", pathVar="petId") int q2) {

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

}
```

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

```
// GET /pets/42

@GetMapping("/pets/{petId}")
public void findPet(@MatrixVariable(required=false, 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

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable MultiValueMap<String, String> matrixVars,
        @MatrixVariable(pathVar="petId"") MultiValueMap<String, String> petMatrixVars)
{

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

The MVC Java config and the MVC namespace both provide options for enabling the use of matrix variables.

If you are using Java config, The Advanced Customizations with MVC Java Config section describes how the `RequestMappingHandlerMapping` can be customized.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven enable-matrix-variables="true"/>

</beans>
```

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:

```
@PostMapping(path = "/pets", <strong>consumes = "application/json"</strong>)
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`. Also consider using constants provided in `MediaType` such as `APPLICATION_JSON_VALUE` and `APPLICATION_JSON_UTF8_VALUE`.

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:

```
@GetMapping(path = "/pets/{petId}", <strong>produces = MediaType
.APPLICATION_JSON_UTF8_VALUE</strong>)
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model) {
    // implementation omitted
}
```

Be aware that the media type specified in the *produces* condition can also optionally specify a character set. For example, in the code snippet above we specify the same media type than the default one configured in `MappingJackson2HttpMessageConverter`, including the `UTF-8` charset.

Just like with *consumes*, producible media type expressions can be negated as in `!text/plain` to match to all requests other than those with an `Accept` header value of `text/plain`. Also consider using constants provided in `MediaType` such as `APPLICATION_JSON_VALUE` and `APPLICATION_JSON_UTF8_VALUE`.

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 {

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

    @GetMapping(path = "/pets", <strong>headers = "myHeader=myValue"</strong>)
    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.

HTTP HEAD and HTTP OPTIONS

`@RequestMapping` methods mapped to "GET" are also implicitly mapped to "HEAD", i.e. there is no need to have "HEAD" explicitly declared. An HTTP HEAD request is processed as if it were an HTTP GET except instead of writing the body only the number of bytes are counted and the "Content-Length" header set.

`@RequestMapping` methods have built-in support for HTTP OPTIONS. By default an HTTP OPTIONS request is handled by setting the "Allow" response header to the HTTP methods explicitly declared on all `@RequestMapping` methods with matching URL patterns. When no HTTP methods are explicitly declared the "Allow" header is set to "GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS". Ideally always declare the HTTP method(s) that an `@RequestMapping` method is intended to handle, or alternatively use one of the dedicated *composed* `@RequestMapping` variants (see Composed @RequestMapping Variants).

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

1.3.3. Defining @RequestMapping handler methods

`@RequestMapping` handler methods can have 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 being `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:

- `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 API.
- Request or response objects (Servlet API). Choose any specific request or response type, for example `ServletRequest` or `HttpServletRequest` or Spring's `MultipartRequest` /`MultipartHttpServletRequest`.
- 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.

- `java.servlet.http.PushBuilder` for the associated Servlet 4.0 push builder API, allowing for programmatic HTTP/2 resource pushes.
- `java.security.Principal` (or a specific `Principal` implementation class if known), containing the currently authenticated user.
- `org.springframework.http.HttpMethod` for the HTTP request method, represented as Spring's `HttpMethod` enum.
- `java.util.Locale` for the current request locale, determined by the most specific locale resolver available, in effect, the configured `LocaleResolver` / `LocaleContextResolver` in an MVC environment.
- `java.util.TimeZone` (Java 6+) / `java.time.ZoneId` (Java 8+) for the time zone associated with the current request, as determined by a `LocaleContextResolver`.

- `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.
- `@PathVariable` annotated parameters for access to URI template variables. See URI Template Patterns.
- `@MatrixVariable` annotated parameters for access to name-value pairs located in URI path segments. See Matrix Variables.
- `@RequestParam` annotated parameters for access to specific Servlet request parameters. Parameter values are converted to the declared method argument type. See 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. See Mapping request header attributes with the @RequestHeader annotation.
- `@RequestBody` annotated parameters for access to the HTTP request body. Parameter values are converted to the declared method argument type using `HttpMessageConverter`s. See Mapping the request body with the @RequestBody annotation.
- `@RequestPart` annotated parameters for access to the content of a "multipart/form-data" request part. See Handling a file upload request from programmatic clients and Multipart (file upload) support.
- `@SessionAttribute` annotated parameters for access to existing, permanent session attributes (e.g. user authentication object) as opposed to model attributes temporarily stored in the session as part of a controller workflow via `@SessionAttributes`.
- `@RequestAttribute` annotated parameters for access to request attributes.
- `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 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). See Passing Data To the Redirect Target and Using flash attributes.
- 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 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 than one model object and Spring will create a separate `BindingResult` instance for each of them so the following sample won't work:

Invalid ordering of BindingResult and @ModelAttribute

```
@PostMapping
public String processSubmit(<strong>@ModelAttribute("pet") Pet pet</strong>, Model
model, <strong>BindingResult result</strong>) { ... }
```

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

```
@PostMapping
public String processSubmit(<strong>@ModelAttribute("pet") Pet pet</strong>, <strong
>BindingResult result</strong>, Model model) { ... }
```

JDK 1.8's `java.util.Optional` is supported as a method parameter type with annotations that have a `required` attribute (e.g. `@RequestParam`, `@RequestHeader`, etc). The use of `java.util.Optional` in those cases is equivalent to having `required=false`.

Supported method return types

The following are the supported return types:

- `ModelAndView` object (Spring MVC), providing a view, model attributes, and optionally a response status.
- `Rendering` object (Spring WebFlux), providing a view, model attributes, and optionally a response status.
- `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.
- `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.
- `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).

- `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 Mapping the response body with the @ResponseBody annotation.
- `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 Using HttpEntity.
- `HttpHeaders` object to return a response with no body.
- `Callable<?>` async computation in a Spring MVC managed thread.
- `DeferredResult<?>` async result produced later from an application managed, or any thread.
- `ListenableFuture<?>` as an alternative equivalent to using `DeferredResult`.
- `CompletableFuture<?>` or `CompletionStage<?>` as an alternative equivalent to `DeferredResult`.
- `ResponseBodyEmitter` can be returned to write multiple objects to the response asynchronously; also supported as the body within a `ResponseEntity`.
- `SseEmitter` can be returned to write Server-Sent Events to the response asynchronously; also supported as the body within a `ResponseEntity`.
- `StreamingResponseBody` can be returned to write to the response OutputStream asynchronously; also supported as the body within a `ResponseEntity`.
- Reactive types from Reactor 3, RxJava 2, RxJava 1 or others registered through the configured `ReactiveAdapterRegistry` can be returned as an alternative equivalent to using `DeferredResult` for single-valued types, or `ResponseBodyEmitter` and `SseEmitter` for multi-valued reactive types where a streaming media type (e.g. "text/event-stream", "application/json+stream") is requested.
- 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 {

    // ...

    @GetMapping
    public String setupForm(<strong>@RequestParam("petId") int petId</strong>,
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(name="id", required=false)`).

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

When an `@RequestParam` annotation is used on a `Map<String, String>` or `MultiValueMap<String, String>` argument, the map is populated with all request parameters.

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:

```
@PutMapping("/something")
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.RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <util:list id="beanList">
            <ref bean="stringHttpMessageConverter"/>
            <ref bean="marshallingHttpMessageConverter"/>
        </util:list>
    </property>
</bean>

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

<bean id="marshallingHttpMessageConverter"
        class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter
">
    <property name="marshaller" ref="castorMarshaller"/>
    <property name="unmarshaller" ref="castorMarshaller"/>
</bean>

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

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 placed 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:

```
@GetMapping("/something")
@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.

Creating REST Controllers with the @RestController annotation

It's a very common use case to have Controllers implement a REST API, thus serving only JSON, XML or custom MediaType content. For convenience, instead of annotating all your `@RequestMapping` methods with `@ResponseBody`, you can annotate your controller Class with `@RestController`.

`@RestController` is a stereotype annotation that combines `@ResponseBody` and `@Controller`. More than that, it gives more meaning to your Controller and also may carry additional semantics in future releases of the framework.

As with regular `@Controller`s, a `@RestController` may be assisted by `@ControllerAdvice` or `@RestControllerAdvice` beans. See the Advising controllers with @ControllerAdvice and @RestControllerAdvice section for more details.

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 many controllers. See the Advising controllers with @ControllerAdvice and @RestControllerAdvice section for more details.

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 then derived based on view name conventions instead, much like for methods returning `void` — see 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.

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(<strong>@ModelAttribute Pet pet</strong>) { }
```

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 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 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:

```
@PutMapping("/accounts/{account}")
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 Validation. Customizing the data binding process for a controller level is covered in 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:

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(<strong>@ModelAttribute("pet") Pet pet</strong>,
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.

Note that in some cases it may be useful to gain access to an attribute in the model without data binding. For such cases you may inject the `Model` into the controller or alternatively use the `binding` flag on the annotation:

```
@ModelAttribute
public AccountForm setUpForm() {
    return new AccountForm();
}

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

@PostMapping("update")
public String update(@Valid AccountUpdateForm form, BindingResult result,
        <strong>@ModelAttribute(binding=false)</strong> Account account) {

    // ...
}
```

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:

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(<strong>@ModelAttribute("pet") Pet pet</strong>,
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:

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(<strong>@Valid @ModelAttribute("pet") Pet pet</strong>,
BindingResult result) {

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

    // ...

}
```

See Bean validation and Spring validation 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")
<strong>@SessionAttributes("pet")</strong>
public class EditPetForm {
    // ...
}
```

Using @SessionAttribute to access pre-existing global session attributes

If you need access to pre-existing session attributes that are managed globally, i.e. outside the controller (e.g. by a filter), and may or may not be present use the `@SessionAttribute` annotation on a method parameter:

```
@RequestMapping("/")
public String handle(<strong>@SessionAttribute</strong> User user) {
    // ...
}
```

For use cases that require adding or removing session attributes consider injecting `org.springframework.web.context.request.WebRequest` or `javax.servlet.http.HttpSession` into the controller method.

For temporary storage of model attributes in the session as part of a controller workflow consider using `SessionAttributes` as described in Using @SessionAttributes to store model attributes in the HTTP session between requests.

Using @RequestAttribute to access request attributes

Similar to `@SessionAttribute` the `@RequestAttribute` annotation can be used to access pre-existing request attributes created by a filter or interceptor:

```
@RequestMapping("/")
public String handle(<strong>@RequestAttribute</strong> Client client) {
    // ...
}
```

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(<strong>@CookieValue("JSESSIONID")</strong> String
cookie) {
    //...
}
```

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

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(<strong>@RequestHeader("Accept-Encoding")</strong>
String encoding,
        <strong>@RequestHeader("Keep-Alive")</strong> long keepAlive) {
    //...
}
```

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

When an `@RequestHeader` annotation is used on a `Map<String, String>`, `MultiValueMap<String, String>`, or `HttpHeaders` argument, the map is populated with all header values.

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>`.

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 Customizing WebDataBinder initialization) or by registering `Formatters` with the `FormattingConversionService` (see Spring 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`. See the Advising controllers with @ControllerAdvice and @RestControllerAdvice section for more details.

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` methods support, 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 {

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

    // ...
}
```

Alternatively, as of Spring 4.2, consider using `addCustomFormatter` to specify `Formatter` implementations instead of `PropertyEditor` instances. This is particularly useful if you happen to have a `Formatter`-based setup in a shared `FormattingConversionService` as well, with the same

approach to be reused for controller-specific tweaking of the binding rules.

```
@Controller
public class MyFormController {

    <strong>@InitBinder</strong>
    protected void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
    }

    // ...
}
```

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 `RequestMappingHandlerAdapter`, 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.RequestMappingHandlerAdapter">
    <property name="cacheSeconds" value="0"/>
    <property name="webBindingInitializer">
        <bean class=
"org.springframework.samples.petclinic.web.ClinicBindingInitializer"/>
    </property>
</bean>
```

`@InitBinder` methods can also be defined in an `@ControllerAdvice`-annotated class in which case they apply to matching controllers. This provides an alternative to using a `WebBindingInitializer`. See the Advising controllers with @ControllerAdvice and @RestControllerAdvice section for more details.

Advising controllers with @ControllerAdvice and @RestControllerAdvice

The `@ControllerAdvice` annotation is a component annotation allowing implementation classes to be auto-detected through classpath scanning. It is automatically enabled when using the MVC namespace or the MVC Java config.

Classes annotated with `@ControllerAdvice` can contain `@ExceptionHandler`, `@InitBinder`, and `@ModelAttribute` annotated methods, and these methods will apply to `@RequestMapping` methods across all controller hierarchies as opposed to the controller hierarchy within which they are declared.

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

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

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class AnnotationAdvice {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class BasePackageAdvice {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController
.class})
public class AssignableTypesAdvice {}
```

Check out the `@ControllerAdvice` documentation for more details.

Jackson Serialization View Support

It can sometimes be useful to filter contextually the object that will be serialized to the HTTP response body. In order to provide such capability, Spring MVC has built-in support for rendering with Jackson's Serialization Views.

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

```
@RestController
public class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView.class)
    public User getUser() {
        return new User("eric", "7!jd#h23");
    }
}

public class User {

    public interface WithoutPasswordView {};
    public interface WithPasswordView extends WithoutPasswordView {};

    private String username;
    private String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    @JsonView(WithoutPasswordView.class)
    public String getUsername() {
        return this.username;
    }

    @JsonView(WithPasswordView.class)
    public String getPassword() {
        return this.password;
    }
}
```

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

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

```
@Controller
public class UserController extends AbstractController {

    @GetMapping("/user")
    public String getUser(Model model) {
        model.addAttribute("user", new User("eric", "7!jd#h23"));
        model.addAttribute(JsonView.class.getName(), User.WithoutPasswordView.class);
        return "userView";
    }
}
```

Jackson JSONP Support

In order to enable JSONP support for `@ResponseBody` and `ResponseEntity` methods, declare an `@ControllerAdvice` bean that extends `AbstractJsonpResponseBodyAdvice` as shown below where the constructor argument indicates the JSONP query parameter name(s):

```
@ControllerAdvice
public class JsonpAdvice extends AbstractJsonpResponseBodyAdvice {

    public JsonpAdvice() {
        super("callback");
    }
}
```

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

1.3.4. Asynchronous Request Processing

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

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

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

}
```

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

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

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

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

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

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

- Controller returns a `Callable`.

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

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

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

For further background on the motivation for async request processing and when or why to use it please read this blog post series.

Exception Handling for Async Requests

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

Intercepting Async Requests

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

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

The `DeferredResult` type also provides methods such as `onTimeout(Runnable)` and

`onCompletion(Runnable)`. See the Javadoc of `DeferredResult` for more details.

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

HTTP Streaming

A controller method can use `DeferredResult` and `Callable` to produce its return value asynchronously and that can be used to implement techniques such as long polling where the server can push an event to the client as soon as possible.

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

Here is an example of that:

```
@RequestMapping("/events")
public ResponseBodyEmitter handle() {
    ResponseBodyEmitter emitter = new ResponseBodyEmitter();
    // Save the emitter somewhere..
    return emitter;
}

// In some other thread
emitter.send("Hello once");

// and again later on
emitter.send("Hello again");

// and done at some point
emitter.complete();
```

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

HTTP Streaming With Server-Sent Events

`SseEmitter` is a sub-class of `ResponseBodyEmitter` providing support for Server-Sent Events. Server-sent events is a just another variation on the same "HTTP Streaming" technique except events pushed from the server are formatted according to the W3C Server-Sent Events specification.

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

Note however that Internet Explorer does not support Server-Sent Events and that for more advanced web application messaging scenarios such as online games, collaboration, financial

applicatinos, and others it's better to consider Spring's WebSocket support that includes SockJS-style WebSocket emulation falling back to a very wide range of browsers (including Internet Explorer) and also higher-level messaging patterns for interacting with clients through a publish-subscribe model within a more messaging-centric architecture. For further background on this see the following blog post.

HTTP Streaming Directly To The OutputStream

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

Here is an example of that:

```
@RequestMapping("/download")
public StreamingResponseBody handle() {
    return new StreamingResponseBody() {
        @Override
        public void writeTo(OutputStream outputStream) throws IOException {
            // write...
        }
    };
}
```

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

Async Requests with Reactive Types

If using the reactive `WebClient` from `spring-webflux`, or another client, or a data store with reactive support, you can return reactive types directly from Spring MVC controller methods.

- If the return type has single-value stream semantics such as Reactor `Mono` or RxJava `Single` it is adapted and equivalent to using `DeferredResult`.
- If the return type has multi-value stream semantics such as Reactor `Flux` or RxJava `Observable` / `Flowable` and if the media type indicates streaming, e.g. "application/stream+json" or "text/event-stream", it is adapted and equivalent to using `ResponseBodyEmitter` or `SseEmitter`. You can also return `Flux<ServerSentEvent>` or `Observable<ServerSentEvent>`.
- If the return type has multi-value stream semantics but the media type does not imply streaming, e.g. "application/json", it is adapted and equivalent to using `DeferredResult<List<?>>`, e.g. JSON array.

Reactive libraries are detected and adapted to a Reactive Streams `Publisher` through Spring's pluggable `ReactiveAdapterRegistry` which by default supports Reactor 3, RxJava 2, and RxJava 1. Note that for RxJava 1 you will need to add "io.reactivex:rxjava-reactive-streams" to the classpath.

A common assumption with reactive libraries is to not block the processing thread. The `WebClient` with Reactor Netty for example is based on event-loop style handling using a small, fixed number of threads and those must not be blocked when writing to the `ServletResponseOutputStream`. Reactive libraries have operators for that but Spring MVC automatically writes asynchronously so you don't need to use them. The underlying `TaskExecutor` for this must be configured through the MVC Java config and the MVC namespace as described in the following section which by default is a `SyncTaskExecutor` and hence not suitable for production use.

Unlike Spring MVC, Spring WebFlux is built on a non-blocking, reactive foundation and uses the Servlet 3.1 non-blocking I/O that's also based on event loop style processing and hence does not require a thread to absorb the effect of blocking.

Configuring Asynchronous Request Processing

Servlet Container Configuration

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

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
            http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    ...

</web-app>
```

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

Below is some example web.xml configuration:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
            http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <filter>
        <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
        <filter-class>org.springframework.~.OpenEntityManagerInViewFilter</filter-
class>
        <async-supported>true</async-supported>
    </filter>

    <filter-mapping>
        <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
        <url-pattern>/*</url-pattern>
        <dispatcher>REQUEST</dispatcher>
        <dispatcher>ASYNC</dispatcher>
    </filter-mapping>

</web-app>
```

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

Spring MVC Configuration

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

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

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

1.3.5. Testing Controllers

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

1.4. 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 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 so you cannot use `urlDecode=false` with prefix-based Servlet mappings and likewise must also set `alwaysUseFullPath=true`.

The following example shows how to configure an interceptor:

```
<beans>
    <bean class=
"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">
        <property name="interceptors">
            <bean class="example.MyInterceptor"/>
        </property>
    </bean>
</beans>
```

1.4.1. Intercepting requests with a HandlerInterceptor

Spring's handler mapping mechanism includes handler interceptors, which are useful when you

want to apply specific functionality to certain requests, for example, checking for a principal.

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

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

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

```
<beans>
    <bean id="handlerMapping"
            class=
"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="officeHoursInterceptor"/>
            </list>
        </property>
    </bean>

    <bean id="officeHoursInterceptor"
            class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime" value="9"/>
        <property name="closingTime" value="18"/>
    </bean>
</beans>
```

```
package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;

    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }

    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }

    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
            Object handler) throws Exception {
        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        if (openingTime <= hour && hour < closingTime) {
            return true;
        }
        response.sendRedirect("http://host.com/outsideOfficeHours.html");
        return false;
    }
}
```

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

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.

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

1.5. Resolving views

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

1.5.1. Resolving views with the ViewResolver interface

As discussed in 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 3. View resolvers

ViewResolver	Description
`AbstractCachingViewResolver`	Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching.
`XmlViewResolver`	Implementation of `ViewResolver` that accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is `/WEB-INF/views.xml`.
`ResourceBundleViewResolver`	Implementation of `ViewResolver` that uses bean definitions in a `ResourceBundle`, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is `views.properties`.
`UrlBasedViewResolver`	Simple implementation of the `ViewResolver` interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.

ViewResolver	Description
`InternalResourceViewResolver`	Convenient subclass of `UrlBasedViewResolver` that supports `InternalResourceView` (in effect, Servlets and JSPs) and subclasses such as `JstlView` and `TilesView`. You can specify the view class for all views generated by this resolver by using `setViewClass(..)`. See the `UrlBasedViewResolver` javadocs for details.
`FreeMarkerViewResolver`	Convenient subclass of `UrlBasedViewResolver` that supports `FreeMarkerView` and custom subclasses of them.
`ContentNegotiatingViewResolver`	Implementation of the `ViewResolver` interface that resolves a view based on the request file name or `Accept` header. See 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 FreeMarker template is modified), you can use the `removeFromCache(String viewName, Locale loc)` method.

1.5.2. Chaining ViewResolvers

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

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

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

<bean id="excelViewResolver" class=
"org.springframework.web.servlet.view.XmlViewResolver">
    <property name="order" value="1"/>
    <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in views.xml -->

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

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

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

the chain not being fully inspected, because the `InternalResourceViewResolver` will *always* return a view!

1.5.3. Redirecting to Views

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

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

RedirectView

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

If you use `RedirectView` and the view is created by the controller itself, it is recommended that you configure the redirect URL to be injected into the controller so that it is not baked into the controller but configured in the context along with the view names. The The redirect: prefix facilitates this decoupling.

Passing Data To the Redirect Target

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

Appending primitive type attributes as query parameters may be the desired result if a model instance was prepared specifically for the redirect. However, in annotated controllers the model may contain additional attributes added for rendering purposes (e.g. drop-down field values). To avoid the possibility of having such attributes appear in the URL, an `@RequestMapping` method can

declare an argument of type `RedirectAttributes` and use it to specify the exact attributes to make available to `RedirectView`. If the method does redirect, the content of `RedirectAttributes` is used. Otherwise the content of the model is used.

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

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

```
@PostMapping("/files/{path}")
public String upload(...) {
    // ...
    return "redirect:files/{path}";
}
```

Another way of passing data to the redirect target is via *Flash Attributes*. Unlike other redirect attributes, flash attributes are saved in the HTTP session (and hence do not appear in the URL). See Using flash attributes for more information.

The redirect: prefix

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

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

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

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

The forward: prefix

It is also possible to use a special `forward:` prefix for view names that are ultimately resolved by `UrlBasedViewResolver` and subclasses. This creates an `InternalResourceView` (which ultimately does a

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

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

1.5.4. 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 custom resolution of a view based on a file extension, use a `ContentNegotiationManager`: see Content Negotiation.

Here is an example configuration of a `ContentNegotiatingViewResolver`:

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="viewResolvers">
        <list>
            <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
            <bean class=
"org.springframework.web.servlet.view.InternalResourceViewResolver">
                <property name="prefix" value="/WEB-INF/jsp/"/>
                <property name="suffix" value=".jsp"/>
            </bean>
        </list>
    </property>
    <property name="defaultViews">
        <list>
            <bean class=
"org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
        </list>
    </property>
</bean>

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

The `InternalResourceViewResolver` handles the translation of view names and JSP pages, while the `BeanNameViewResolver` returns a view based on the name of a bean. (See "Resolving views with the ViewResolver interface" 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>();

    @GetMapping("/content")
    public ModelAndView getContent() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("content");
        mav.addObject("sampleContentList", contentList);
        return mav;
    }

}
```

1.6. 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 .

1.7. Building URIs

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

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

```
UriComponents uriComponents = UriComponentsBuilder.fromUriString(
        "http://example.com/hotels/{hotel}/bookings/{booking}").build();

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

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

You can also expand and encode using individual URI components:

```
UriComponents uriComponents = UriComponentsBuilder.newInstance()
        .scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}"
).build()
        .expand("42", "21")
        .encode();
```

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

```
HttpServletRequest request = ...

// Re-use host, scheme, port, path and query string
// Replace the "accountId" query param

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)
        .replaceQueryParam("accountId", "{id}").build()
        .expand("123")
        .encode();
```

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

```
// Re-use host, port and context path
// Append "/accounts" to the path

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(request)
        .path("/accounts").build()
```

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

```
// Re-use host, port, context path
// Append the literal part of the servlet mapping to the path
// Append "/accounts" to the path

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping
(request)
        .path("/accounts").build()
```

1.7.1. Building URIs to Controllers and methods

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

```
@Controller
@RequestMapping("/hotels/{hotel}")
public class BookingController {

    @GetMapping("/bookings/{booking}")
    public String getBooking(@PathVariable Long booking) {

    // ...
       }
}
```

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

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

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

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

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

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

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

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

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

```
UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path
("/en");
MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);
builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

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

1.7.2. Working with "Forwarded" and "X-Forwarded-*" Headers

As a request goes through proxies such as load balancers the host, port, and scheme may change presenting a challenge for applications that need to create links to resources since the links should reflect the host, port, and scheme of the original request as seen from a client perspective.

RFC 7239 defines the "Forwarded" HTTP header for proxies to use to provide information about the original request. There are also other non-standard headers in use such as "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto".

Both `ServletUriComponentsBuilder` and `MvcUriComponentsBuilder` detect, extract, and use information from the "Forwarded" header, or from "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto" if "Forwarded" is not present, so that the resulting links reflect the original request.

The `ForwardedHeaderFilter` provides an alternative to do the same once and globally for the entire application. The filter wraps the request in order to overlay host, port, and scheme information and also "hides" any forwarded headers for subsequent processing.

Note that there are security considerations when using forwarded headers as explained in Section 8 of RFC 7239. At the application level it is difficult to determine whether forwarded headers can be trusted or not. This is why the network upstream should be configured correctly to filter out untrusted forwarded headers from the outside.

Applications that don't have a proxy and don't need to use forwarded headers can configure the `ForwardedHeaderFilter` to remove and ignore such headers.

1.7.3. Building URIs to Controllers and methods from views

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

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

The assigned request mapping names are logged at TRACE level on startup.

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

For example given:

```
@RequestMapping("/people/{id}/addresses")
public class PersonAddressController {

    @RequestMapping("/{country}")
    public HttpEntity getAddress(@PathVariable String country) { ... }
}
```

You can prepare a link from a JSP as follows:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
...
<a href="${s:mvcUrl('PAC#getAddress').arg(0,'US').buildAndExpand('123')}">Get
Address</a>
```

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

1.8. 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 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.

1.8.1. Obtaining Time Zone Information

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

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

1.8.2. AcceptHeaderLocaleResolver

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

1.8.3. CookieLocaleResolver

This locale resolver inspects a `Cookie` that might exist on the client to see if a `Locale` or `TimeZone` is

specified. If so, it uses the specified details. Using the properties of this locale resolver, you can specify the name of the cookie as well as the maximum age. Find below an example of defining a `CookieLocaleResolver`.

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

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

    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser
shuts down) -->
    <property name="cookieMaxAge" value="100000"/>

</bean>
```

Table 4. CookieLocaleResolver properties

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

1.8.4. SessionLocaleResolver

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

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

1.8.5. LocaleChangeInterceptor

You can enable changing of locales by adding the `LocaleChangeInterceptor` to one of the handler mappings (see 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>
```

1.9. Using themes

1.9.1. 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.

1.9.2. 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.

1.9.3. 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 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.

1.10. Multipart (file upload) support

1.10.1. 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.

1.10.2. 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.

1.10.3. 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>
```

1.10.4. 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 {

    @PostMapping("/form")
    public String handleFormUpload(@RequestParam("name") String name,
            @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }

        return "redirect:uploadFailure";
    }

}
```

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

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

```
@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(@RequestParam("name") String name,
            @RequestParam("file") Part file) {

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

        return "redirect:uploadSuccess";
    }

}
```

1.10.5. Handling a file upload request from programmatic clients

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

```
POST /someUrl
Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="meta-data"
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit

{
    "name": "value"
}
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="file-data"; filename="file.properties"
Content-Type: text/xml
Content-Transfer-Encoding: 8bit
... File Data ...
```

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

You can use the `@RequestPart` annotation instead of the `@RequestParam` annotation for this purpose. It allows you to have the content of a specific multipart passed through an `HttpMessageConverter`

taking into consideration the `'Content-Type'` header of the multipart:

```
@PostMapping("/someUrl")
public String onSubmit(<strong>@RequestPart("meta-data") MetaData metadata,
        @RequestPart("file-data") MultipartFile file</strong>) {

    // ...

}
```

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`.

1.11. Handling exceptions

1.11.1. HandlerExceptionResolver

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

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

1.11.2. @ExceptionHandler

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

You can do that with `@ExceptionHandler` methods. When declared within a controller such methods

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

```
@Controller
public class SimpleController {

    // @RequestMapping methods omitted ...

    @ExceptionHandler(IOException.class)
    public ResponseEntity<String> handleIOException(IOException ex) {
        // prepare responseEntity
        return responseEntity;
    }

}
```

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

Much like standard controller methods annotated with a `@RequestMapping` annotation, the method arguments and return values of `@ExceptionHandler` methods can be flexible. For example, the `HttpServletRequest` can be accessed in Servlet environments. 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.

1.11.3. Handling Standard Spring MVC Exceptions

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

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

Exception	HTTP Status Code
`BindException`	400 (Bad Request)
`ConversionNotSupportedException`	500 (Internal Server Error)
`HttpMediaTypeNotAcceptableException`	406 (Not Acceptable)

Exception	HTTP Status Code
`HttpMediaTypeNotSupportedException`	415 (Unsupported Media Type)
`HttpMessageNotReadableException`	400 (Bad Request)
`HttpMessageNotWritableException`	500 (Internal Server Error)
`HttpRequestMethodNotSupportedException`	405 (Method Not Allowed)
`MethodArgumentNotValidException`	400 (Bad Request)
`MissingPathVariableException`	500 (Internal Server Error)
`MissingServletRequestParameterException`	400 (Bad Request)
`MissingServletRequestPartException`	400 (Bad Request)
`NoHandlerFoundException`	404 (Not Found)
`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`, `MappingJackson2JsonView`, and so on. However, you may prefer to use `@ExceptionHandler` methods instead.

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

1.11.4. REST Controller Exception Handling

An `@RestController` may use `@ExceptionHandler` methods that return a `ResponseEntity` to provide both a response status and error details in the body of the response. Such methods may also be added to `@ControllerAdvice` classes for exception handling across a subset or all controllers.

A common requirement is to include error details in the body of the response. Spring does not automatically do this (although Spring Boot does) because the representation of error details in the response body is application specific.

Applications that wish to implement a global exception handling strategy with error details in the response body should consider extending the abstract base class `ResponseEntityExceptionHandler` which provides handling for the exceptions that Spring MVC raises and provides hooks to customize the response body as well as to handle other exceptions. Simply declare the extension class as a Spring bean and annotate it with `@ControllerAdvice`. For more details see See `ResponseEntityExceptionHandler`.

1.11.5. 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.

1.11.6. Customizing the Default Servlet Container Error Page

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

```
<error-page>
    <location>/error</location>
</error-page>
```

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

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

```
@Controller
public class ErrorController {

    @RequestMapping(path = "/error", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ResponseBody
    public Map<String, Object> handle(HttpServletRequest request) {

        Map<String, Object> map = new HashMap<String, Object>();
        map.put("status", request.getAttribute("javax.servlet.error.status_code"));
        map.put("reason", request.getAttribute("javax.servlet.error.message"));

        return map;
    }

}
```

or in a JSP:

```
<%@ page contentType="application/json" pageEncoding="UTF-8"%>
{
    status:<%=request.getAttribute("javax.servlet.error.status_code") %>,
    reason:<%=request.getAttribute("javax.servlet.error.message") %>
}
```

1.12. Web Security

The Spring Security project provides features to protect web applications from malicious exploits. Check out the reference documentation in the sections on "CSRF protection", "Security Response Headers", and also "Spring MVC Integration". Note that using Spring Security to secure the application is not necessarily required for all features. For example CSRF protection can be added simply by adding the `CsrfFilter` and `CsrfRequestDataValueProcessor` to your configuration. See the Spring MVC Showcase for an example.

Another option is to use a framework dedicated to Web Security. HDIV is one such framework and integrates with Spring MVC.

1.13. Convention over configuration support

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

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

1.13.1. The Controller ControllerClassNameHandlerMapping

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

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

```
public class <strong>ViewShoppingCartController</strong> implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response) {
        // the implementation is not hugely important for this example...
    }

}
```

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

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

<bean id="<strong>viewShoppingCart</strong>" class="x.y.z.ViewShoppingCartController">
    <!-- inject dependencies as required... -->
</bean>
```

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

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

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

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

- `AdminController` maps to the `/admin/*` request URL
- `CatalogController` maps to the `/catalog/*` request URL

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

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

1.13.2. The Model ModelMap (ModelAndView)

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

```
public class DisplayShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response) {

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

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

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

        return mav;
    }
}
```

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

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

What, no automatic pluralization?

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

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

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

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

1.13.3. The View - RequestToViewNameTranslator

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

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

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response) {
        // process the request...
        ModelAndView mav = new ModelAndView();
        // add data as necessary to the model...
        return mav;
        // notice that no View or logical view name has been set
    }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- this bean with the well known name generates view names for us -->
    <bean id="viewNameTranslator"
            class=
"org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator"/>

    <bean class="x.y.RegistrationController">
        <!-- inject dependencies as necessary -->
    </bean>

    <!-- maps request URLs to Controller names -->
    <bean class=
"org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

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

</beans>
```

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

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

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

1.14. HTTP caching support

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

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

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

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

1.14.1. Cache-Control HTTP header

Spring Web MVC supports many use cases and ways to configure "Cache-Control" headers for an application. While RFC 7234 Section 5.2.2 completely describes that header and its possible directives, there are several ways to address the most common cases.

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

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

The `CacheControl` builder class simply describes the available "Cache-Control" directives and makes it easier to build your own HTTP caching strategy. Once built, a `CacheControl` instance can then be accepted as an argument in several Spring Web MVC APIs.

```
// Cache for an hour - "Cache-Control: max-age=3600"
   CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

   // Prevent caching - "Cache-Control: no-store"
   CacheControl ccNoStore = CacheControl.noStore();

   // Cache for ten days in public and private caches,
   // public caches should not transform the response
   // "Cache-Control: max-age=864000, public, no-transform"
   CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS)
                                       .noTransform().cachePublic();
```

1.14.2. HTTP caching support for static resources

Static resources should be served with appropriate `'Cache-Control'` and conditional headers for optimal performance. Configuring a `ResourceHttpRequestHandler` for serving static resources not only natively writes `'Last-Modified'` headers by reading a file's metadata, but also `'Cache-Control'` headers if properly configured.

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

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public-resources/")
                .setCacheControl(CacheControl.maxAge(1, TimeUnit.HOURS).cachePublic()
);
    }

}
```

And in XML:

```
<mvc:resources mapping="/resources/**" location="/public-resources/">
    <mvc:cache-control max-age="3600" cache-public="true"/>
</mvc:resources>
```

1.14.3. Support for the Cache-Control, ETag and Last-Modified response headers in Controllers

Controllers can support `'Cache-Control'`, `'ETag'`, and/or `'If-Modified-Since'` HTTP requests; this is

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

As described in Using HttpEntity, controllers can interact with the request/response using `HttpEntity` types. Controllers returning `ResponseEntity` can include HTTP caching information in responses like this:

```
@GetMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

    Book book = findBook(id);
    String version = book.getVersion();

    return ResponseEntity
                .ok()
                .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
                .eTag(version) // lastModified is also available
                .body(book);
}
```

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

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

```
@RequestMapping
public String myHandleMethod(WebRequest webRequest, Model model) {

    long lastModified = // 1. application-specific calculation

    if (request.checkNotModified(lastModified)) {
        // 2. shortcut exit - no further processing necessary
        return null;
    }

    // 3. or otherwise further request processing, actually preparing content
    model.addAttribute(...);
    return "myViewName";
}
```

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

Note that there are 3 variants for this:

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

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

1.14.4. Shallow ETag support

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

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

This filter has a `writeWeakETag` parameter that configures the filter to write Weak ETags, like this: `W/"02a2d595e6ed9a0b24f027f2b63b134d6"`, as defined in RFC 7232 Section 2.3.

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

```
<filter>
    <filter-name>etagFilter</filter-name>
    <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-
class>
    <!-- Optional parameter that configures the filter to write weak ETags
    <init-param>
        <param-name>writeWeakETag</param-name>
        <param-value>true</param-value>
    </init-param>
    -->
</filter>

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

Or in Servlet 3.0+ environments,

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] { new ShallowEtagHeaderFilter() };
    }

}
```

See Code-based Servlet container initialization for more details.

1.15. Code-based Servlet container initialization

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

```
import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext = new XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

        ServletRegistration.Dynamic registration = container.addServlet("dispatcher",
new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }

}
```

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

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

```
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

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

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

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

}
```

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

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext cxt = new XmlWebApplicationContext();
        cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return cxt;
    }

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

}
```

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

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] { new HiddenHttpMethodFilter(), new
CharacterEncodingFilter() };
    }

}
```

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

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

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

1.16. Configuring Spring MVC

Special Bean Types In the WebApplicationContext and 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 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.

1.16.1. Enabling the MVC Java Config or the MVC XML Namespace

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

```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven/>

</beans>
```

The above registers a RequestMappingHandlerMapping, a RequestMappingHandlerAdapter, and an ExceptionHandlerExceptionResolver (among others) in support of processing requests with

annotated controller methods using annotations such as `@RequestMapping`, `@ExceptionHandler`, and others.

It also enables the following:

1. Spring 3 style type conversion through a ConversionService 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:

   a. `ByteArrayHttpMessageConverter` converts byte arrays.
   b. `StringHttpMessageConverter` converts strings.
   c. `ResourceHttpMessageConverter` converts to/from `org.springframework.core.io.Resource` for all media types.
   d. `SourceHttpMessageConverter` converts to/from a `javax.xml.transform.Source`.
   e. `FormHttpMessageConverter` converts form data to/from a `MultiValueMap<String, String>`.
   f. `Jaxb2RootElementHttpMessageConverter` converts Java objects to/from XML — added if JAXB2 is present and Jackson 2 XML extension is not present on the classpath.
   g. `MappingJackson2HttpMessageConverter` converts to/from JSON — added if Jackson 2 is present on the classpath.
   h. `MappingJackson2XmlHttpMessageConverter` converts to/from XML — added if Jackson 2 XML extension is present on the classpath.
   i. `MappingJackson2SmileHttpMessageConverter` converts to/from Smile (binary JSON) — added if Jackson 2 Smile extension is present on the classpath.
   j. `MappingJackson2CborHttpMessageConverter` converts to/from CBOR — added if Jackson 2 CBOR extension is present on the classpath.
   k. `AtomFeedHttpMessageConverter` converts Atom feeds — added if Rome is present on the classpath.
   l. `RssChannelHttpMessageConverter` converts RSS feeds — added if Rome is present on the classpath.

See Message Converters for more information about how to customize these default converters.

Jackson JSON and XML converters are created using `ObjectMapper` instances created by `Jackson2ObjectMapperBuilder` in order to provide a better default configuration.

This builder customizes Jackson's default properties with the following ones:

1. `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled.
2. `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled.

It also automatically registers the following well-known modules if they are detected on the classpath:

1. jackson-datatype-jdk7: support for Java 7 types like `java.nio.file.Path`.
2. jackson-datatype-joda: support for Joda-Time types.
3. jackson-datatype-jsr310: support for Java 8 Date & Time API types.
4. jackson-datatype-jdk8: support for other Java 8 types like `Optional`.

1.16.2. Customizing the Provided Configuration

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

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    // Override configuration methods...

}
```

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

1.16.3. Conversion and Formatting

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

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

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

}
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

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

    <bean id="conversionService"
            class=
"org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="org.example.MyConverter"/>
            </set>
        </property>
        <property name="formatters">
            <set>
                <bean class="org.example.MyFormatter"/>
                <bean class="org.example.MyAnnotationFormatterFactory"/>
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean class="org.example.MyFormatterRegistrar"/>
            </set>
        </property>
    </bean>

</beans>
```

See FormatterRegistrar SPI and the FormattingConversionServiceFactoryBean for more information on when to use FormatterRegistrars.

1.16.4. Validation

Spring provides a Validator interface that can be used for validation in all layers of an application. In Spring MVC you can configure it for use as a global Validator instance, to be used whenever an @Valid or @Validated controller method argument is encountered, and/or as a local Validator within a controller through an @InitBinder method. Global and local validator instances can be combined to provide composite validation.

Spring also supports JSR-303/JSR-349 Bean Validation via LocalValidatorFactoryBean which adapts the Spring org.springframework.validation.Validator interface to the Bean Validation javax.validation.Validator contract. This class can be plugged into Spring MVC as a global validator as described next.

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

Sometimes it's convenient to have a LocalValidatorFactoryBean injected into a controller or another class. The easiest way to do that is to declare your own @Bean and also mark it with @Primary in order to avoid a conflict with the one provided with the MVC Java config.

If you prefer to use the one from the MVC Java config, you'll need to override the mvcValidator method from WebMvcConfigurationSupport and declare the method to explicitly return LocalValidatorFactory rather than Validator. See Advanced Customizations with MVC Java Config for information on how to switch to extend the provided configuration.

Alternatively you can configure your own global Validator instance:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public Validator getValidator(); {
        // return "global" validator
    }

}
```

and in XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

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

</beans>
```

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

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

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

1.16.5. Interceptors

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

An example of registering interceptors in Java:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleInterceptor());
        registry.addInterceptor(new ThemeInterceptor()).addPathPatterns("/**")
.excludePathPatterns("/admin/**");
        registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/secure/*
");
    }

}
```

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

```
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <mvc:exclude-mapping path="/admin/**"/>
        <bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/secure/*"/>
        <bean class="org.example.SecurityInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

1.16.6. Content Negotiation

You can configure how Spring MVC determines the requested media types from the request. The available options are to check a query parameter, the URL path for a file extension, the "Accept" header, use a fixed list, or a custom strategy.

By default for backwards compatibility the path extension in the request URI is checked first and the "Accept" header is checked second. However if you must use URL-based content type resolution, we highly recommend using the query parameter strategy over the path extension since the latter can cause issues with URI variables, path parameters, and also in combination with URI decoding.

The MVC Java config and the MVC namespace register `json`, `xml`, `rss`, `atom` by default if corresponding dependencies are on the classpath. Additional path extension-to-media type mappings may also be registered explicitly and that also has the effect of whitelisting them as safe extensions for the purpose of RFD attack detection (see Suffix Pattern Matching and RFD for more detail).

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

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.mediaType("json", MediaType.APPLICATION_JSON);
    }
}
```

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

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

<bean id="contentNegotiationManager" class=
"org.springframework.web.accept.ContentNegotiationManagerFactoryBean">
    <property name="mediaTypes">
        <value>
            json=application/json
            xml=application/xml
        </value>
    </property>
</bean>
```

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

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

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

1.16.7. View Controllers

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

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

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @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"/>
```

1.16.8. View Resolvers

The MVC config simplifies the registration of view resolvers.

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

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.jsp();
    }

}
```

And the same in XML:

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class=
"org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:jsp/>
</mvc:view-resolvers>
```

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

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

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class=
"org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:freemarker cache="false"/>
</mvc:view-resolvers>

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

In Java config simply add the respective "Configurer" bean:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.freeMarker().cache(false);
    }

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("/WEB-INF/");
        return configurer;
    }

}
```

1.16.9. Serving of Resources

This option allows static resource requests following a particular URL pattern to be served by a `ResourceHttpRequestHandler` from any of a list of `Resource` locations. This provides a convenient way to serve static resources from locations other than the web application root, including locations on the classpath. The `cache-period` property may be used to set far future expiration headers (1 year is the recommendation of optimization tools such as Page Speed and YSlow) so that they will be more efficiently utilized by the client. The handler also properly evaluates the `Last-Modified` header (if

present) so that a `304` status code will be returned as appropriate, avoiding unnecessary overhead for resources that are already cached by the client. For example, to serve resource requests with a URL pattern of `/resources/**` from a `public-resources` directory within the web application root you would use:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @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 implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**").addResourceLocations("/public-
resources/").setCachePeriod(31556926);
    }

}
```

And in XML:

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

For more details, see HTTP caching support for static resources.

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 implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/", "classpath:/META-INF/public-web-resources/
");
    }

}
```

And in XML:

```
<mvc:resources mapping="/resources/**" location="/, classpath:/META-INF/public-web-
resources/"/>
```

When serving resources that may change when a new version of the application is deployed it is recommended that you incorporate a version string into the mapping pattern used to request the resources so that you may force clients to request the newly deployed version of your application's resources. Support for versioned URLs is built into the framework and can be enabled by configuring a resource chain on the resource handler. The chain consists of one more `ResourceResolver` instances followed by one or more `ResourceTransformer` instances. Together they can provide arbitrary resolution and transformation of resources.

The built-in `VersionResourceResolver` can be configured with different strategies. For example a `FixedVersionStrategy` can use a property, a date, or other as the version. A `ContentVersionStrategy` uses an MD5 hash computed from the content of the resource (known as "fingerprinting" URLs). Note that the `VersionResourceResolver` will automatically use the resolved version strings as HTTP ETag header values when serving resources.

`ContentVersionStrategy` is a good default choice to use except in cases where it cannot be used (e.g. with JavaScript module loaders). You can configure different version strategies against different patterns as shown below. Keep in mind also that computing content-based versions is expensive and therefore resource chain caching should be enabled in production.

Java config example;

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public-resources/")
                .resourceChain(true).addResolver(
                    new VersionResourceResolver().addContentVersionStrategy("/**"));
    }

}
```

XML example:

```
<mvc:resources mapping="/resources/**" location="/public-resources/">
    <mvc:resource-chain>
        <mvc:resource-cache/>
        <mvc:resolvers>
            <mvc:version-resolver>
                <mvc:content-version-strategy patterns="/**"/>
            </mvc:version-resolver>
        </mvc:resolvers>
    </mvc:resource-chain>
</mvc:resources>
```

In order for the above to work the application must also render URLs with versions. The easiest way to do that is to configure the `ResourceUrlEncodingFilter` which wraps the response and overrides its `encodeURL` method. This will work in JSPs, FreeMarker, and any other view technology that calls the response `encodeURL` method. Alternatively, an application can also inject and use directly the `ResourceUrlProvider` bean, which is automatically declared with the MVC Java config and the MVC namespace.

Webjars are also supported with `WebJarsResourceResolver`, which is automatically registered when the `"org.webjars:webjars-locator"` library is on classpath. This resolver allows the resource chain to resolve version agnostic libraries from HTTP GET requests `"GET /jquery/jquery.min.js"` will return resource `"/jquery/1.2.0/jquery.min.js"`. It also works by rewriting resource URLs in templates `<script src="/jquery/jquery.min.js"/> → <script src="/jquery/1.2.0/jquery.min.js"/>`.

1.16.10. Falling Back On the "Default" Servlet To Serve Resources

This allows for mapping the `DispatcherServlet` to "/" (thus overriding the mapping of the container's default Servlet), while still allowing static resource requests to be handled by the container's default Servlet. It configures a `DefaultServletHttpRequestHandler` with a URL mapping of "/**" and the lowest priority relative to other URL mappings.

This handler will forward all requests to the default Servlet. Therefore it is important that it

remains last in the order of all other URL `HandlerMappings`. That will be the case if you use `<mvc:annotation-driven>` or alternatively if you are setting up your own customized `HandlerMapping` instance be sure to set its `order` property to a value lower than that of the `DefaultServletHttpRequestHandler`, which is `Integer.MAX_VALUE`.

To enable the feature using the default setup use:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @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 implements WebMvcConfigurer {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable("myCustomDefaultServlet");
    }

}
```

Or in XML:

```
<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>
```

1.16.11. Path Matching

This allows customizing various settings related to URL mapping and path matching. For details on the individual options check out the PathMatchConfigurer API.

Below is an example in Java config:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer
            .setUseSuffixPatternMatch(true)
            .setUseTrailingSlashMatch(false)
            .setUseRegisteredSuffixPatternMatch(true)
            .setPathMatcher(antPathMatcher())
            .setUrlPathHelper(urlPathHelper());
    }

    @Bean
    public UrlPathHelper urlPathHelper() {
        //...
    }

    @Bean
    public PathMatcher antPathMatcher() {
        //...
    }

}
```

And the same in XML, use the `<mvc:path-matching>` element:

```
<mvc:annotation-driven>
    <mvc:path-matching
        suffix-pattern="true"
        trailing-slash="false"
        registered-suffixes-only="true"
        path-helper="pathHelper"
        path-matcher="pathMatcher"/>
</mvc:annotation-driven>

<bean id="pathHelper" class="org.example.app.MyPathHelper"/>
<bean id="pathMatcher" class="org.example.app.MyPathMatcher"/>
```

1.16.12. Message Converters

Customization of `HttpMessageConverter` can be achieved in Java config by overriding `configureMessageConverters()` if you want to replace the default converters created by Spring MVC, or by overriding `extendMessageConverters()` if you just want to customize them or add additional converters to the default ones.

Below is an example that adds Jackson JSON and XML converters with a customized `ObjectMapper` instead of default ones:

```
@Configuration
@EnableWebMvc
public class WebConfiguration implements WebMvcConfigurer {

    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()
                .indentOutput(true)
                .dateFormat(new SimpleDateFormat("yyyy-MM-dd"))
                .modulesToInstall(new ParameterNamesModule());
        converters.add(new MappingJackson2HttpMessageConverter(builder.build()));
        converters.add(new MappingJackson2XmlHttpMessageConverter(builder.xml().build
()));
    }

}
```

In this example, `Jackson2ObjectMapperBuilder` is used to create a common configuration for both `MappingJackson2HttpMessageConverter` and `MappingJackson2XmlHttpMessageConverter` with indentation enabled, a customized date format and the registration of jackson-module-parameter-names that adds support for accessing parameter names (feature added in Java 8).

Enabling indentation with Jackson XML support requires `woodstox-core-asl` dependency in addition to `jackson-dataformat-xml` one.

Other interesting Jackson modules are available:

1. jackson-datatype-money: support for `javax.money` types (unofficial module)
2. jackson-datatype-hibernate: support for Hibernate specific types and properties (including lazy-loading aspects)

It is also possible to do the same in XML:

```
<mvc:annotation-driven>
    <mvc:message-converters>
        <bean class=
"org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
            <property name="objectMapper" ref="objectMapper"/>
        </bean>
        <bean class=
"org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter">
            <property name="objectMapper" ref="xmlMapper"/>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="objectMapper" class=
"org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean"
      p:indentOutput="true"
      p:simpleDateFormat="yyyy-MM-dd"
      p:modulesToInstall="
com.fasterxml.jackson.module.paramnames.ParameterNamesModule"/>

<bean id="xmlMapper" parent="objectMapper" p:createXmlMapper="true"/>
```

1.16.13. Advanced Customizations with MVC Java Config

As you can see from the above examples, MVC Java config and the MVC namespace provide higher level constructs that do not require deep knowledge of the underlying beans created for you. Instead it helps you to focus on your application needs. However, at some point you may need more fine-grained control or you may simply wish to understand the underlying configuration.

The first step towards more fine-grained control is to see the underlying beans created for you. In MVC Java config you can see the javadocs and the `@Bean` methods in `WebMvcConfigurationSupport`. The configuration in this class is automatically imported through the `@EnableWebMvc` annotation. In fact if you open `@EnableWebMvc` you can see the `@Import` statement.

The next step towards more fine-grained control is to customize a property on one of the beans created in `WebMvcConfigurationSupport` or perhaps to provide your own instance. This requires two things — remove the `@EnableWebMvc` annotation in order to prevent the import and then extend from `DelegatingWebMvcConfiguration`, a subclass of `WebMvcConfigurationSupport`. Here is an example:

```
@Configuration
public class WebConfig extends DelegatingWebMvcConfiguration {

    @Override
    public void addInterceptors(InterceptorRegistry registry){
        // ...
    }

    @Override
    @Bean
    public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {
        // Create or let "super" create the adapter
        // Then customize one of its properties
    }

}
```

An application should have only one configuration extending `DelegatingWebMvcConfiguration` or a single `@EnableWebMvc` annotated class, since they both register the same underlying beans.

Modifying beans in this way does not prevent you from using any of the higher-level constructs shown earlier in this section. `WebMvcConfigurer` subclasses and `WebMvcConfigurer` implementations are still being used.

1.16.14. Advanced Customizations with the MVC Namespace

Fine-grained control over the configuration created for you is a bit harder with the MVC namespace.

If you do need to do that, rather than replicating the configuration it provides, consider configuring a `BeanPostProcessor` that detects the bean you want to customize by type and then modifying its properties as necessary. For example:

```
@Component
public class MyPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String name) throws
BeansException {
        if (bean instanceof RequestMappingHandlerAdapter) {
            // Modify properties of the adapter
        }
    }

}
```

Note that `MyPostProcessor` needs to be included in an `<component scan/>` in order for it to be detected

or if you prefer you can declare it explicitly with an XML bean declaration.

Chapter 2. Spring MVC View Technologies

2.1. Introduction

One of the areas in which Spring excels is in the separation of view technologies from the rest of the MVC framework. For example, deciding to use Groovy Markup Templates or Thymeleaf in place of an existing JSP is primarily a matter of configuration. This chapter covers the major view technologies that work with Spring and touches briefly on how to add new ones. This chapter assumes you are already familiar with Resolving views which covers the basics of how views in general are coupled to the MVC framework.

2.2. Thymeleaf

Thymeleaf is a good example of a view technology fitting perfectly in the MVC framework. Support for this integration is not provided by the Spring team but by the Thymeleaf team itself.

Configuring Thymeleaf for Spring usually requires a few beans defined, like a `ServletContextTemplateResolver`, a `SpringTemplateEngine` and a `ThymeleafViewResolver`. Please refer to the Thymeleaf+Spring documentation section for more details.

2.3. Groovy Markup Templates

The Groovy Markup Template Engine is another view technology, supported by Spring. This template engine is a template engine primarily aimed at generating XML-like markup (XML, XHTML, HTML5, ...), but that can be used to generate any text based content.

This requires Groovy 2.3.1+ on the classpath.

2.3.1. Configuration

Configuring the Groovy Markup Template Engine is quite easy:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

       @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.groovy();
    }

    @Bean
    public GroovyMarkupConfigurer groovyMarkupConfigurer() {
        GroovyMarkupConfigurer configurer = new GroovyMarkupConfigurer();
        configurer.setResourceLoaderPath("/WEB-INF/");
        return configurer;
    }
}
```

The XML counterpart using the MVC namespace is:

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:groovy/>
</mvc:view-resolvers>

<mvc:groovy-configurer resource-loader-path="/WEB-INF/"/>
```

2.3.2. Example

Unlike traditional template engines, this one relies on a DSL that uses the builder syntax. Here is a sample template for an HTML page:

```
yieldUnescaped '<!DOCTYPE html>'
html(lang:'en') {
    head {
        meta('http-equiv':'"Content-Type" content="text/html; charset=utf-8"')
        title('My page')
    }
    body {
        p('This is an example of HTML contents')
    }
}
```

2.4. FreeMarker

FreeMarker is a templating language that can be used as a view technology within Spring MVC

applications. For details on the template language, see the FreeMarker web site.

2.4.1. Dependencies

Your web application will need to include `freemarker-2.x.jar` in order to work with FreeMarker. Typically this is included in the `WEB-INF/lib` folder where the jars are guaranteed to be found by a Java EE server and added to the classpath for your application. It is of course assumed that you already have the `spring-webmvc.jar` in your `'WEB-INF/lib'` directory too!

2.4.2. Context configuration

A suitable configuration is initialized by adding the relevant configurer bean definition to your `'*-servlet.xml'` as shown below:

```
<!-- freemarker config -->
<bean id="freemarkerConfig" class=
"org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>

<!--
View resolvers can also be configured with ResourceBundles or XML files. If you need
different view resolving based on Locale, you have to use the resource bundle
resolver.
-->
<bean id="viewResolver" class=
"org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
    <property name="cache" value="true"/>
    <property name="prefix" value=""/>
    <property name="suffix" value=".ftl"/>
</bean>
```

For non web-apps add a `FreeMarkerConfigurationFactoryBean` to your application context definition file.

2.4.3. Creating templates

Your templates need to be stored in the directory specified by the `FreeMarkerConfigurer` shown above. If you use the view resolvers highlighted, then the logical view names relate to the template file names in similar fashion to `InternalResourceViewResolver` for JSP's. So if your controller returns a ModelAndView object containing a view name of "welcome" then the resolver will look for the `/WEB-INF/freemarker/welcome.ftl` template.

2.4.4. Advanced FreeMarker configuration

FreeMarker 'Settings' and 'SharedVariables' can be passed directly to the FreeMarker `Configuration` object managed by Spring by setting the appropriate bean properties on the `FreeMarkerConfigurer` bean. The `freemarkerSettings` property requires a `java.util.Properties` object and the

`freemarkerVariables` property requires a `java.util.Map`.

```
<bean id="freemarkerConfig" class=
"org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
    <property name="freemarkerVariables">
        <map>
            <entry key="xml_escape" value-ref="fmXmlEscape"/>
        </map>
    </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

See the FreeMarker documentation for details of settings and variables as they apply to the `Configuration` object.

2.4.5. Bind support and form handling

Spring provides a tag library for use in JSP's that contains (amongst other things) a `<spring:bind/>` tag. This tag primarily enables forms to display values from form backing objects and to show the results of failed validations from a `Validator` in the web or business tier. Spring also has support for the same functionality in FreeMarker, with additional convenience macros for generating form input elements themselves.

The bind macros

A standard set of macros are maintained within the `spring-webmvc.jar` file for both languages, so they are always available to a suitably configured application.

Some of the macros defined in the Spring libraries are considered internal (private) but no such scoping exists in the macro definitions making all macros visible to calling code and user templates. The following sections concentrate only on the macros you need to be directly calling from within your templates. If you wish to view the macro code directly, the file is called `spring.ftl` in the package `org.springframework.web.servlet.view.freemarker`.

Simple binding

In your HTML forms (vm / ftl templates) which act as a form view for a Spring MVC controller, you can use code similar to the following to bind to field values and display error messages for each input field in similar fashion to the JSP equivalent. Example code is shown below for the `personForm` view configured earlier:

```
<!-- freemarker macros have to be imported into a namespace. We strongly
recommend sticking to 'spring' -->
<#import "/spring.ftl" as spring/>
<html>
    ...
    <form action="" method="POST">
        Name:
        <@spring.bind "myModelObject.name"/>
        <input type="text"
            name="${spring.status.expression}"
            value="${spring.status.value?html}"/><br>
        <#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>
        <br>
        ...
        <input type="submit" value="submit"/>
    </form>
    ...
</html>
```

`<@spring.bind>` requires a 'path' argument which consists of the name of your command object (it will be 'command' unless you changed it in your FormController properties) followed by a period and the name of the field on the command object you wish to bind to. Nested fields can be used too such as "command.address.street". The `bind` macro assumes the default HTML escaping behavior specified by the ServletContext parameter `defaultHtmlEscape` in `web.xml`.

The optional form of the macro called `<@spring.bindEscaped>` takes a second argument and explicitly specifies whether HTML escaping should be used in the status error messages or values. Set to true or false as required. Additional form handling macros simplify the use of HTML escaping and these macros should be used wherever possible. They are explained in the next section.

Form input generation macros

Additional convenience macros for both languages simplify both binding and form generation (including validation error display). It is never necessary to use these macros to generate form input fields, and they can be mixed and matched with simple HTML or calls direct to the spring bind macros highlighted previously.

The following table of available macros show the VTL and FTL definitions and the parameter list that each takes.

Table 6. Table of macro definitions

macro	FTL definition	message (output a string from a resource bundle based on the code parameter)
<@spring.message code/>	**messageText** (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter)	<@spring.messageText code, text/>
url (prefix a relative URL with the application's context root)	<@spring.url relativeUrl/>	**formInput** (standard input field for gathering user input)
<@spring.formInput path, attributes, fieldType/>	**formHiddenInput** * (hidden input field for submitting non-user input)	<@spring.formHiddenInput path, attributes/>
formPasswordInput * (standard input field for gathering passwords. Note that no value will ever be populated in fields of this type)	<@spring.formPasswordInput path, attributes/>	**formTextarea** (large text field for gathering long, freeform text input)
<@spring.formTextarea path, attributes/>	**formSingleSelect** (drop down box of options allowing a single required value to be selected)	<@spring.formSingleSelect path, options, attributes/>
formMultiSelect (a list box of options allowing the user to select 0 or more values)	<@spring.formMultiSelect path, options, attributes/>	**formRadioButtons** (a set of radio buttons allowing a single selection to be made from the available choices)
<@spring.formRadioButtons path, options separator, attributes/>	**formCheckboxes** (a set of checkboxes allowing 0 or more values to be selected)	<@spring.formCheckboxes path, options, separator, attributes/>

macro	FTL definition	message (output a string from a resource bundle based on the code parameter)
formCheckbox (a single checkbox)	<@spring.formCheckbox path, attributes/>	**showErrors** (simplify display of validation errors for the bound field)

- In FTL (FreeMarker), these two macros are not actually required as you can use the normal `formInput` macro, specifying `' hidden' or ' 'password' as the value for the 'fieldType` parameter.

The parameters to any of the above macros have consistent meanings:

- path: the name of the field to bind to (ie "command.name")
- options: a Map of all the available values that can be selected from in the input field. The keys to the map represent the values that will be POSTed back from the form and bound to the command object. Map objects stored against the keys are the labels displayed on the form to the user and may be different from the corresponding values posted back by the form. Usually such a map is supplied as reference data by the controller. Any Map implementation can be used depending on required behavior. For strictly sorted maps, a `SortedMap` such as a `TreeMap` with a suitable Comparator may be used and for arbitrary Maps that should return values in insertion order, use a `LinkedHashMap` or a `LinkedMap` from commons-collections.
- separator: where multiple options are available as discreet elements (radio buttons or checkboxes), the sequence of characters used to separate each one in the list (ie "
").
- attributes: an additional string of arbitrary tags or text to be included within the HTML tag itself. This string is echoed literally by the macro. For example, in a textarea field you may supply attributes as 'rows="5" cols="60"' or you could pass style information such as 'style="border:1px solid silver"'.
- classOrStyle: for the showErrors macro, the name of the CSS class that the span tag wrapping each error will use. If no information is supplied (or the value is empty) then the errors will be wrapped in tags.

Examples of the macros are outlined below some in FTL and some in VTL. Where usage differences exist between the two languages, they are explained in the notes.

Input Fields

The formInput macro takes the path parameter (command.name) and an additional attributes parameter which is empty in the example above. The macro, along with all other form generation macros, performs an implicit spring bind on the path parameter. The binding remains valid until a new bind occurs so the showErrors macro doesn't need to pass the path parameter again - it simply operates on whichever field a bind was last created for.

The showErrors macro takes a separator parameter (the characters that will be used to separate

multiple errors on a given field) and also accepts a second parameter, this time a class name or style attribute. Note that FreeMarker is able to specify default values for the attributes parameter.

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

Output is shown below of the form fragment generating the name field, and displaying a validation error after the form was submitted with no value in the field. Validation occurs through Spring's Validation framework.

The generated HTML looks like this:

```
Name:
<input type="text" name="name" value="">
<br>
    <b>required</b>
<br>
<br>
```

The formTextarea macro works the same way as the formInput macro and accepts the same parameter list. Commonly, the second parameter (attributes) will be used to pass style information or rows and cols attributes for the textarea.

Selection Fields

Four selection field macros can be used to generate common UI value selection inputs in your HTML forms.

- formSingleSelect
- formMultiSelect
- formRadioButtons
- formCheckboxes

Each of the four macros accepts a Map of options containing the value for the form field, and the label corresponding to that value. The value and the label can be the same.

An example of radio buttons in FTL is below. The form backing object specifies a default value of 'London' for this field and so no validation is necessary. When the form is rendered, the entire list of cities to choose from is supplied as reference data in the model under the name 'cityMap'.

```
...
Town:
<@spring.formRadioButtons "command.address.town", cityMap, ""/><br><br>
```

This renders a line of radio buttons, one for each value in `cityMap` using the separator "". No additional attributes are supplied (the last parameter to the macro is missing). The cityMap uses the

same String for each key-value pair in the map. The map's keys are what the form actually submits as POSTed request parameters, map values are the labels that the user sees. In the example above, given a list of three well known cities and a default value in the form backing object, the HTML would be

```
Town:
<input type="radio" name="address.town" value="London">London</input>
<input type="radio" name="address.town" value="Paris" checked="checked">Paris</input>
<input type="radio" name="address.town" value="New York">New York</input>
```

If your application expects to handle cities by internal codes for example, the map of codes would be created with suitable keys like the example below.

```
protected Map<String, String> referenceData(HttpServletRequest request) throws
Exception {
    Map<String, String> cityMap = new LinkedHashMap<>();
    cityMap.put("LDN", "London");
    cityMap.put("PRS", "Paris");
    cityMap.put("NYC", "New York");

    Map<String, String> model = new HashMap<>();
    model.put("cityMap", cityMap);
    return model;
}
```

The code would now produce output where the radio values are the relevant codes but the user still sees the more user friendly city names.

```
Town:
<input type="radio" name="address.town" value="LDN">London</input>
<input type="radio" name="address.town" value="PRS" checked="checked">Paris</input>
<input type="radio" name="address.town" value="NYC">New York</input>
```

HTML escaping and XHTML compliance

Default usage of the form macros above will result in HTML tags that are HTML 4.01 compliant and that use the default value for HTML escaping defined in your web.xml as used by Spring's bind support. In order to make the tags XHTML compliant or to override the default HTML escaping value, you can specify two variables in your template (or in your model where they will be visible to your templates). The advantage of specifying them in the templates is that they can be changed to different values later in the template processing to provide different behavior for different fields in your form.

To switch to XHTML compliance for your tags, specify a value of 'true' for a model/context variable named xhtmlCompliant:

```
<#-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>
```

Any tags generated by the Spring macros will now be XHTML compliant after processing this directive.

In similar fashion, HTML escaping can be specified per field:

```
<-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true in spring>
<-- next field will use HTML escaping -->
<@spring.formInput "command.name"/>

<assign htmlEscape = false in spring>
<-- all future fields will be bound with HTML escaping off -->
```

2.5. JSP & JSTL

Spring provides a couple of out-of-the-box solutions for JSP and JSTL views. Using JSP or JSTL is done using a normal view resolver defined in the `WebApplicationContext`. Furthermore, of course you need to write some JSPs that will actually render the view.

Setting up your application to use JSTL is a common source of error, mainly caused by confusion over the different servlet spec., JSP and JSTL version numbers, what they mean and how to declare the taglibs correctly. The article How to Reference and Use JSTL in your Web Application provides a useful guide to the common pitfalls and how to avoid them. Note that as of Spring 3.0, the minimum supported servlet version is 2.4 (JSP 2.0 and JSTL 1.1), which reduces the scope for confusion somewhat.

2.5.1. View resolvers

Just as with any other view technology you're integrating with Spring, for JSPs you'll need a view resolver that will resolve your views. The most commonly used view resolvers when developing with JSPs are the `InternalResourceViewResolver` and the `ResourceBundleViewResolver`. Both are declared in the `WebApplicationContext`:

```
<!-- the ResourceBundleViewResolver -->
<bean id="viewResolver" class=
"org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
</bean>

# And a sample properties file is uses (views.properties in WEB-INF/classes):
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

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

As you can see, the `ResourceBundleViewResolver` needs a properties file defining the view names mapped to 1) a class and 2) a URL. With a `ResourceBundleViewResolver` you can mix different types of views using only one resolver.

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

The `InternalResourceBundleViewResolver` can be configured for using JSPs as described above. As a best practice, we strongly encourage placing your JSP files in a directory under the `'WEB-INF'` directory, so there can be no direct access by clients.

2.5.2. 'Plain-old' JSPs versus JSTL

When using the Java Standard Tag Library you must use a special view class, the `JstlView`, as JSTL needs some preparation before things such as the I18N features will work.

2.5.3. Additional tags facilitating development

Spring provides data binding of request parameters to command objects as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a few tags that make things even easier. All Spring tags have*HTML escaping* features to enable or disable escaping of characters.

The tag library descriptor (TLD) is included in the `spring-webmvc.jar`. Further information about the individual tags can be found in the appendix entitled [spring.tld].

2.5.4. Using Spring's form tag library

As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC. Each tag provides support for the set of

attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use. The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

Unlike other form/input tag libraries, Spring's form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with. As you will see in the following examples, the form tags make JSPs easier to develop, read and maintain.

Let's go through the form tags and look at an example of how each tag is used. We have included generated HTML snippets where certain tags require further commentary.

Configuration

The form tag library comes bundled in `spring-webmvc.jar`. The library descriptor is called `spring-form.tld`.

To use the tags from this library, add the following directive to the top of your JSP page:

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

where `form` is the tag name prefix you want to use for the tags from this library.

The form tag

This tag renders an HTML 'form' tag and exposes a binding path to inner tags for binding. It puts the command object in the `PageContext` so that the command object can be accessed by inner tags. *All the other tags in this library are nested tags of the* `form` *tag.*

Let's assume we have a domain object called `User`. It is a JavaBean with properties such as `firstName` and `lastName`. We will use it as the form backing object of our form controller which returns `form.jsp`. Below is an example of what `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

The `firstName` and `lastName` values are retrieved from the command object placed in the `PageContext` by the page controller. Keep reading to see more complex examples of how inner tags are used with the `form` tag.

The generated HTML looks like a standard form:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value="Harry"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value="Potter"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

The preceding JSP assumes that the variable name of the form backing object is `'command'`. If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable like so:

```
<form:form modelAttribute="user">
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

The input tag

This tag renders an HTML 'input' tag using the bound value and type='text' by default. For an example of this tag, see The form tag. Starting with Spring 3.1 you can use other types such HTML5-specific types like 'email', 'tel', 'date', and others.

The checkbox tag

This tag renders an HTML 'input' tag with type 'checkbox'.

Let's assume our `User` has preferences such as newsletter subscription and a list of hobbies. Below is an example of the `Preferences` class:

```
public class Preferences {

    private boolean receiveNewsletter;
    private String[] interests;
    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

The `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --%>
            <td><form:checkbox path="preferences.receiveNewsletter"/></td>
        </tr>

        <tr>
            <td>Interests:</td>
            <%-- Approach 2: Property is of an array or of type java.util.Collection
--%>
            <td>
                Quidditch: <form:checkbox path="preferences.interests" value=
"Quidditch"/>
                Herbology: <form:checkbox path="preferences.interests" value=
"Herbology"/>
                Defence Against the Dark Arts: <form:checkbox path=
"preferences.interests" value="Defence Against the Dark Arts"/>
            </td>
        </tr>

        <tr>
            <td>Favourite Word:</td>
            <%-- Approach 3: Property is of type java.lang.Object --%>
            <td>
                Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>
            </td>
        </tr>
    </table>
</form:form>
```

There are 3 approaches to the `checkbox` tag which should meet all your checkbox needs.

- Approach One - When the bound value is of type `java.lang.Boolean`, the `input(checkbox)` is marked as 'checked' if the bound value is `true`. The `value` attribute corresponds to the resolved value of the `setValue(Object)` value property.
- Approach Two - When the bound value is of type `array` or `java.util.Collection`, the `input(checkbox)` is marked as 'checked' if the configured `setValue(Object)` value is present in the bound `Collection`.
- Approach Three - For any other bound value type, the `input(checkbox)` is marked as 'checked' if the configured `setValue(Object)` is equal to the bound value.

Note that regardless of the approach, the same HTML structure is generated. Below is an HTML snippet of some checkboxes:

```
<tr>
    <td>Interests:</td>
    <td>
        Quidditch: <input name="preferences.interests" type="checkbox" value=
"Quidditch"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Herbology: <input name="preferences.interests" type="checkbox" value=
"Herbology"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Defence Against the Dark Arts: <input name="preferences.interests" type=
"checkbox" value="Defence Against the Dark Arts"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
</tr>
```

What you might not expect to see is the additional hidden field after each checkbox. When a checkbox in an HTML page is *not* checked, its value will not be sent to the server as part of the HTTP request parameters once the form is submitted, so we need a workaround for this quirk in HTML in order for Spring form data binding to work. The `checkbox` tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore ("_") for each checkbox. By doing this, you are effectively telling Spring that "*the checkbox was visible in the form and I want my object to which the form data will be bound to reflect the state of the checkbox no matter what*".

The checkboxes tag

This tag renders multiple HTML 'input' tags with type 'checkbox'.

Building on the example from the previous `checkbox` tag section. Sometimes you prefer not to have to list all the possible hobbies in your JSP page. You would rather provide a list at runtime of the available options and pass that in to the tag. That is the purpose of the `checkboxes` tag. You pass in an `Array`, a `List` or a `Map` containing the available options in the "items" property. Typically the bound property is a collection so it can hold multiple values selected by the user. Below is an example of the JSP using this tag:

```
<form:form>
    <table>
        <tr>
            <td>Interests:</td>
            <td>
                <%-- Property is of an array or of type java.util.Collection --%>
                <form:checkboxes path="preferences.interests" items="${interestList}
"/>
            </td>
        </tr>
    </table>
</form:form>
```

This example assumes that the "interestList" is a `List` available as a model attribute containing

strings of the values to be selected from. In the case where you use a Map, the map entry key will be used as the value and the map entry's value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

The radiobutton tag

This tag renders an HTML 'input' tag with type 'radio'.

A typical usage pattern will involve multiple tag instances bound to the same property but with different values.

```
<tr>
    <td>Sex:</td>
    <td>
        Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/>
    </td>
</tr>
```

The radiobuttons tag

This tag renders multiple HTML 'input' tags with type 'radio'.

Just like the `checkboxes` tag above, you might want to pass in the available options as a runtime variable. For this usage you would use the `radiobuttons` tag. You pass in an `Array`, a `List` or a `Map` containing the available options in the "items" property. In the case where you use a Map, the map entry key will be used as the value and the map entry's value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

```
<tr>
    <td>Sex:</td>
    <td><form:radiobuttons path="sex" items="${sexOptions}"/></td>
</tr>
```

The password tag

This tag renders an HTML 'input' tag with type 'password' using the bound value.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password"/>
    </td>
</tr>
```

Please note that by default, the password value is *not* shown. If you do want the password value to be shown, then set the value of the `'showPassword'` attribute to true, like so.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq" showPassword="true"/>
    </td>
</tr>
```

The select tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested `option` and `options` tags.

Let's assume a `User` has a list of skills.

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
</tr>
```

If the `User's` skill were in Herbology, the HTML source of the 'Skills' row would look like:

```
<tr>
    <td>Skills:</td>
    <td>
        <select name="skills" multiple="true">
            <option value="Potions">Potions</option>
            <option value="Herbology" selected="selected">Herbology</option>
            <option value="Quidditch">Quidditch</option>
        </select>
    </td>
</tr>
```

The option tag

This tag renders an HTML 'option'. It sets 'selected' as appropriate based on the bound value.

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

If the `User`'s house was in Gryffindor, the HTML source of the 'House' row would look like:

```
<tr>
    <td>House:</td>
    <td>
        <select name="house">
            <option value="Gryffindor" selected="selected">Gryffindor</option>
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
</tr>
```

The options tag

This tag renders a list of HTML 'option' tags. It sets the 'selected' attribute as appropriate based on the bound value.

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code" itemLabel="name"/>
        </form:select>
    </td>
</tr>
```

If the `User` lived in the UK, the HTML source of the 'Country' row would look like:

```
<tr>
    <td>Country:</td>
    <td>
        <select name="country">
            <option value="-">--Please Select</option>
            <option value="AT">Austria</option>
            <option value="UK" selected="selected">United Kingdom</option>
            <option value="US">United States</option>
        </select>
    </td>
</tr>
```

As the example shows, the combined usage of an `option` tag with the `options` tag generates the same standard HTML, but allows you to explicitly specify a value in the JSP that is for display only (where it belongs) such as the default string in the example: "-- Please Select".

The `items` attribute is typically populated with a collection or array of item objects. `itemValue` and `itemLabel` simply refer to bean properties of those item objects, if specified; otherwise, the item objects themselves will be stringified. Alternatively, you may specify a `Map` of items, in which case the map keys are interpreted as option values and the map values correspond to option labels. If `itemValue` and/or `itemLabel` happen to be specified as well, the item value property will apply to the map key and the item label property will apply to the map value.

The textarea tag

This tag renders an HTML 'textarea'.

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20"/></td>
    <td><form:errors path="notes"/></td>
</tr>
```

The hidden tag

This tag renders an HTML 'input' tag with type 'hidden' using the bound value. To submit an unbound hidden value, use the HTML `input` tag with type 'hidden'.

```
<form:hidden path="house"/>
```

If we choose to submit the 'house' value as a hidden one, the HTML would look like:

```
<input name="house" type="hidden" value="Gryffindor"/>
```

The errors tag

This tag renders field errors in an HTML 'span' tag. It provides access to the errors created in your controller or those that were created by any validators associated with your controller.

Let's assume we want to display all error messages for the `firstName` and `lastName` fields once we submit the form. We have a validator for instances of the `User` class called `UserValidator`.

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
        return User.class.isAssignableFrom(candidate);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required",
"Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required",
"Field is required.");
    }
}
```

The `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName"/></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

If we submit a form with empty values in the `firstName` and `lastName` fields, this is what the HTML would look like:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

What if we want to display the entire list of errors for a given page? The example below shows that the `errors` tag also supports some basic wildcarding functionality.

- `path="*"` - displays all errors
- `path="lastName"` - displays all errors associated with the `lastName` field
- if `path` is omitted - object errors only are displayed

The example below will display a list of errors at the top of the page, followed by field-specific errors next to the fields:

```
<form:form>
    <form:errors path="*" cssClass="errorBox"/>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
            <td><form:errors path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
            <td><form:errors path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

The HTML would look like:

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is required.<br/>Field is
required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
       </table>
</form>
```

HTTP Method Conversion

A key principle of REST is the use of the Uniform Interface. This means that all resources (URLs) can be manipulated using the same four HTTP methods: GET, PUT, POST, and DELETE. For each method, the HTTP specification defines the exact semantics. For instance, a GET should always be a safe operation, meaning that is has no side effects, and a PUT or DELETE should be idempotent, meaning that you can repeat these operations over and over again, but the end result should be the same. While HTTP defines these four methods, HTML only supports two: GET and POST. Fortunately, there are two possible workarounds: you can either use JavaScript to do your PUT or DELETE, or simply do a POST with the 'real' method as an additional parameter (modeled as a hidden input field in an HTML form). This latter trick is what Spring's `HiddenHttpMethodFilter` does. This filter is a plain Servlet Filter and therefore it can be used in combination with any web framework (not just Spring MVC). Simply add this filter to your web.xml, and a POST with a hidden _method parameter will be converted into the corresponding HTTP method request.

To support HTTP method conversion the Spring MVC form tag was updated to support setting the HTTP method. For example, the following snippet taken from the updated Petclinic sample

```
<form:form method="delete">
    <p class="submit"><input type="submit" value="Delete Pet"/></p>
</form:form>
```

This will actually perform an HTTP POST, with the 'real' DELETE method hidden behind a request parameter, to be picked up by the `HiddenHttpMethodFilter`, as defined in web.xml:

```
<filter>
    <filter-name>httpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>

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

The corresponding `@Controller` method is shown below:

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId, @PathVariable int petId) {
    this.clinic.deletePet(petId);
    return "redirect:/owners/" + ownerId;
}
```

HTML5 Tags

Starting with Spring 3, the Spring form tag library allows entering dynamic attributes, which means you can enter any HTML5 specific attributes.

In Spring 3.1, the form input tag supports entering a type attribute other than 'text'. This is intended to allow rendering new HTML5 specific input types such as 'email', 'date', 'range', and others. Note that entering type='text' is not required since 'text' is the default type.

2.6. Script templates

It is possible to integrate any templating library running on top of a JSR-223 script engine in web applications using Spring. The following describes in a broad way how to do this. The script engine must implement both `ScriptEngine` and `Invocable` interfaces.

It has been tested with:

- Handlebars running on Nashorn
- Mustache running on Nashorn
- React running on Nashorn
- EJS running on Nashorn
- ERB running on JRuby
- String templates running on Jython

2.6.1. Dependencies

To be able to use script templates integration, you need to have available in your classpath the script engine:

- Nashorn Javascript engine is provided builtin with Java 8+. Using the latest update release available is highly recommended.
- Rhino Javascript engine is provided builtin with Java 6 and Java 7. Please notice that using Rhino is not recommended since it does not support running most template engines.
- JRuby dependency should be added in order to get Ruby support.
- Jython dependency should be added in order to get Python support.

You should also need to add dependencies for your script based template engine. For example, for Javascript you can use WebJars to add Maven/Gradle dependencies in order to make your javascript libraries available in the classpath.

2.6.2. How to integrate script based templating

To be able to use script templates, you have to configure it in order to specify various parameters like the script engine to use, the script files to load and what function should be called to render the templates. This is done thanks to a `ScriptTemplateConfigurer` bean and optional script files.

For example, in order to render Mustache templates thanks to the Nashorn Javascript engine provided with Java 8+, you should declare the following configuration:

```
@Configuration
@EnableWebMvc
public class MustacheConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("mustache.js");
        configurer.setRenderObject("Mustache");
        configurer.setRenderFunction("render");
        return configurer;
    }
}
```

The XML counterpart using MVC namespace is:

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:script-template/>
</mvc:view-resolvers>

<mvc:script-template-configurer engine-name="nashorn" render-object="Mustache" render-
function="render">
    <mvc:script location="mustache.js"/>
</mvc:script-template-configurer>
```

The controller is exactly what you should expect:

```
@Controller
public class SampleController {

    @RequestMapping
    public ModelAndView test() {
        ModelAndView mav  = new ModelAndView();
        mav.addObject("title", "Sample title").addObject("body", "Sample body");
        mav.setViewName("template.html");
        return mav;
    }
}
```

And the Mustache template is:

```
<html>
    <head>
        <title>{{title}}</title>
    </head>
    <body>
        <p>{{body}}</p>
    </body>
</html>
```

The render function is called with the following parameters:

- `String template`: the template content
- `Map model`: the view model
- `String url`: the template url (since 4.2.2)

`Mustache.render()` is natively compatible with this signature, so you can call it directly.

If your templating technology requires some customization, you may provide a script that implements a custom render function. For example, Handlerbars needs to compile templates before using them, and requires a polyfill in order to emulate some browser facilities not available in the server-side script engine.

```
@Configuration
@EnableWebMvc
public class MustacheConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("polyfill.js", "handlebars.js", "render.js");
        configurer.setRenderFunction("render");
        configurer.setSharedEngine(false);
        return configurer;
    }
}
```

Setting the `sharedEngine` property to `false` is required when using non thread-safe script engines with templating libraries not designed for concurrency, like Handlebars or React running on Nashorn for example. In that case, Java 8u60 or greater is required due to this bug.

`polyfill.js` only defines the `window` object needed by Handlebars to run properly:

```
var window = {};
```

This basic `render.js` implementation compiles the template before using it. A production ready implementation should also store and reused cached templates / pre-compiled templates. This can be done on the script side, as well as any customization you need (managing template engine configuration for example).

```
function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}
```

Check out Spring script templates unit tests (java, resources) for more configuration examples.

2.7. XML Marshalling View

The `MarshallingView` uses an XML `Marshaller` defined in the `org.springframework.oxm` package to render the response content as XML. The object to be marshalled can be set explicitly using `MarhsallingView's` `` `modelKey `` bean property. Alternatively, the view will iterate over all model properties and marshal the first type that is supported by the `Marshaller`. For more information on the functionality in the `org.springframework.oxm` package refer to the chapter Marshalling XML using O/X Mappers.

2.8. Tiles

It is possible to integrate Tiles - just as any other view technology - in web applications using Spring. The following describes in a broad way how to do this.

This section focuses on Spring's support for Tiles v3 in the `org.springframework.web.servlet.view.tiles3` package.

2.8.1. Dependencies

To be able to use Tiles, you have to add a dependency on Tiles version 3.0.1 or higher and its transitive dependencies to your project.

2.8.2. How to integrate Tiles

To be able to use Tiles, you have to configure it using files containing definitions (for basic information on definitions and other Tiles concepts, please have a look at http://tiles.apache.org). In Spring this is done using the `TilesConfigurer`. Have a look at the following piece of example ApplicationContext configuration:

```
<bean id="tilesConfigurer" class=
"org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>
</bean>
```

As you can see, there are five files containing definitions, which are all located in the `'WEB-INF/defs'` directory. At initialization of the `WebApplicationContext`, the files will be loaded and the definitions factory will be initialized. After that has been done, the Tiles includes in the definition files can be used as views within your Spring web application. To be able to use the views you have to have a `ViewResolver` just as with any other view technology used with Spring. Below you can find two possibilities, the `UrlBasedViewResolver` and the `ResourceBundleViewResolver`.

You can specify locale specific Tiles definitions by adding an underscore and then the locale. For example:

```
<bean id="tilesConfigurer" class=
"org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/tiles.xml</value>
            <value>/WEB-INF/defs/tiles_fr_FR.xml</value>
        </list>
    </property>
</bean>
```

With this configuration, `tiles_fr_FR.xml` will be used for requests with the `fr_FR` locale, and `tiles.xml` will be used by default.

Since underscores are used to indicate locales, it is recommended to avoid using them otherwise in the file names for Tiles definitions.

UrlBasedViewResolver

The `UrlBasedViewResolver` instantiates the given `viewClass` for each view it has to resolve.

```
<bean id="viewResolver" class=
"org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass" value=
"org.springframework.web.servlet.view.tiles3.TilesView"/>
</bean>
```

ResourceBundleViewResolver

The `ResourceBundleViewResolver` has to be provided with a property file containing viewnames and viewclasses the resolver can use:

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

```
...
welcomeView.(class)=org.springframework.web.servlet.view.tiles3.TilesView
welcomeView.url=welcome (this is the name of a Tiles definition)

vetsView.(class)=org.springframework.web.servlet.view.tiles3.TilesView
vetsView.url=vetsView (again, this is the name of a Tiles definition)

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

As you can see, when using the `ResourceBundleViewResolver`, you can easily mix different view technologies.

Note that the `TilesView` class supports JSTL (the JSP Standard Tag Library) out of the box.

SimpleSpringPreparerFactory and SpringBeanPreparerFactory

As an advanced feature, Spring also supports two special Tiles `PreparerFactory` implementations. Check out the Tiles documentation for details on how to use `ViewPreparer` references in your Tiles definition files.

Specify `SimpleSpringPreparerFactory` to autowire ViewPreparer instances based on specified preparer classes, applying Spring's container callbacks as well as applying configured Spring BeanPostProcessors. If Spring's context-wide annotation-config has been activated, annotations in ViewPreparer classes will be automatically detected and applied. Note that this expects preparer *classes* in the Tiles definition files, just like the default `PreparerFactory` does.

Specify `SpringBeanPreparerFactory` to operate on specified preparer *names* instead of classes, obtaining the corresponding Spring bean from the DispatcherServlet's application context. The full bean creation process will be in the control of the Spring application context in this case, allowing for the use of explicit dependency injection configuration, scoped beans etc. Note that you need to define one Spring bean definition per preparer name (as used in your Tiles definitions).

```
<bean id="tilesConfigurer" class=
"org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>

    <!-- resolving preparer names as Spring bean definition names -->
    <property name="preparerFactoryClass"
            value=
"org.springframework.web.servlet.view.tiles3.SpringBeanPreparerFactory"/>

</bean>
```

2.9. XSLT

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML, or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring Web MVC application.

2.9.1. My First Words

This example is a trivial Spring application that creates a list of words in the `Controller` and adds them to the model map. The map is returned along with the view name of our XSLT view. See Implementing Controllers for details of Spring Web MVC's `Controller` interface. The XSLT Controller will turn the list of words into a simple XML document ready for transformation.

Bean definitions

Configuration is standard for a simple Spring application. The MVC configuration has to define a `XsltViewResolver` bean and regular MVC annotation configuration.

```
@EnableWebMvc
@ComponentScan
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public XsltViewResolver xsltViewResolver() {
        XsltViewResolver viewResolver = new XsltViewResolver();
        viewResolver.setPrefix("/WEB-INF/xsl/");
        viewResolver.setSuffix(".xslt");
        return viewResolver;
    }

}
```

And we need a Controller that encapsulates our word generation logic.

Standard MVC controller code

The controller logic is encapsulated in a `@Controller` class, with the handler method being defined like so...

```
@Controller
public class XsltController {

    @RequestMapping("/")
    public String home(Model model) throws Exception {

        Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder()
.newDocument();
        Element root = document.createElement("wordList");

        List<String> words = Arrays.asList("Hello", "Spring", "Framework");
        for (String word : words) {
            Element wordNode = document.createElement("word");
            Text textNode = document.createTextNode(word);
            wordNode.appendChild(textNode);
            root.appendChild(wordNode);
        }

        model.addAttribute("wordList", root);
        return "home";
    }

}
```

So far we've only created a DOM document and added it to the Model map. Note that you can also load an XML file as a `Resource` and use it instead of a custom DOM document.

Of course, there are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

Next, `XsltViewResolver` will resolve the "home" XSLT template file and merge the DOM document into it to generate our view.

Document transformation

Finally, the `XsltViewResolver` will resolve the "home" XSLT template file and merge the DOM document into it to generate our view. As shown in the `XsltViewResolver` configuration, XSLT templates live in the war file in the `'WEB-INF/xsl'` directory and end with a `"xslt"` file extension.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>
                <h1>My First Words</h1>
                <ul>
                    <xsl:apply-templates/>
                </ul>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="word">
        <li><xsl:value-of select="."/></li>
    </xsl:template>

</xsl:stylesheet>
```

This is rendered as:

```
<html>
    <head>
        <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Hello!</title>
    </head>
    <body>
        <h1>My First Words</h1>
        <ul>
            <li>Hello</li>
            <li>Spring</li>
            <li>Framework</li>
        </ul>
    </body>
</html>
```

2.10. Document views (PDF/Excel)

2.10.1. Introduction

Returning an HTML page isn't always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and will be streamed from the server with the correct content type to (hopefully) enable the client PC to run their spreadsheet or PDF viewer application in response.

In order to use Excel views, you need to add the 'poi' library to your classpath, and for PDF generation, the iText library.

2.10.2. Configuration and setup

Document based views are handled in an almost identical fashion to XSLT views, and the following sections build upon the previous one by demonstrating how the same controller used in the XSLT example is invoked to render the same model as both a PDF document and an Excel spreadsheet (which can also be viewed or manipulated in Open Office).

Document view definitions

First, let's amend the views.properties file (or xml equivalent) and add a simple view definition for both document types. The entire file now looks like this with the XSLT view shown from earlier:

```
home.(class)=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.(class)=excel.HomePage

pdf.(class)=pdf.HomePage
```

If you want to start with a template spreadsheet or a fillable PDF form to add your model data to, specify the location as the 'url' property in the view definition

Controller code

The controller code we'll use remains exactly the same from the XSLT example earlier other than to change the name of the view to use. Of course, you could be clever and have this selected based on a URL parameter or some other logic - proof that Spring really is very good at decoupling the views from the controllers!

Subclassing for Excel views

Exactly as we did for the XSLT example, we'll subclass suitable abstract classes in order to implement custom behavior in generating our output documents. For Excel, this involves writing a subclass of `org.springframework.web.servlet.view.document.AbstractExcelView` (for Excel files generated by POI) or `org.springframework.web.servlet.view.document.AbstractJExcelView` (for JExcelApi-generated Excel files) and implementing the `buildExcelDocument()` method.

Here's the complete listing for our POI Excel view which displays the word list from the model map in consecutive rows of the first column of a new spreadsheet:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(Map model, HSSFWorkbook wb, HttpServletRequest
req,
            HttpServletResponse resp) throws Exception {

        HSSFSheet sheet;
        HSSFRow sheetRow;
        HSSFCell cell;

        // Go to the first sheet
        // getSheetAt: only if wb is created from an existing document
        // sheet = wb.getSheetAt(0);
        sheet = wb.createSheet("Spring");
        sheet.setDefaultColumnWidth((short) 12);

        // write a text at A1
        cell = getCell(sheet, 0, 0);
        setText(cell, "Spring-Excel test");

        List words = (List) model.get("wordList");
        for (int i=0; i < words.size(); i++) {
            cell = getCell(sheet, 2+i, 0);
            setText(cell, (String) words.get(i));
        }
    }

}
```

And the following is a view generating the same Excel file, now using JExcelApi:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractJExcelView {

    protected void buildExcelDocument(Map model, WritableWorkbook wb,
            HttpServletRequest request, HttpServletResponse response) throws Exception
{

        WritableSheet sheet = wb.createSheet("Spring", 0);

        sheet.addCell(new Label(0, 0, "Spring-Excel test"));

        List words = (List) model.get("wordList");
        for (int i = 0; i < words.size(); i++) {
            sheet.addCell(new Label(2+i, 0, (String) words.get(i)));
        }
    }
}
```

Note the differences between the APIs. We've found that the JExcelApi is somewhat more intuitive, and furthermore, JExcelApi has slightly better image-handling capabilities. There have been memory problems with large Excel files when using JExcelApi however.

If you now amend the controller such that it returns `xl` as the name of the view (`return new ModelAndView("xl", map);`) and run your application again, you should find that the Excel spreadsheet is created and downloaded automatically when you request the same page as before.

Subclassing for PDF views

The PDF version of the word list is even simpler. This time, the class extends `org.springframework.web.servlet.view.document.AbstractPdfView` and implements the `buildPdfDocument()` method as follows:

```
package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {

    protected void buildPdfDocument(Map model, Document doc, PdfWriter writer,
        HttpServletRequest req, HttpServletResponse resp) throws Exception {
        List words = (List) model.get("wordList");
        for (int i=0; i<words.size(); i++) {
            doc.add( new Paragraph((String) words.get(i)));
        }
    }

}
```

Once again, amend the controller to return the `pdf` view with `return new ModelAndView("pdf", map);`, and reload the URL in your application. This time a PDF document should appear listing each of the words in the model map.

2.11. Feed Views

Both `AbstractAtomFeedView` and `AbstractRssFeedView` inherit from the base class `AbstractFeedView` and are used to provide Atom and RSS Feed views respectfully. They are based on java.net's ROME project and are located in the package `org.springframework.web.servlet.view.feed`.

`AbstractAtomFeedView` requires you to implement the `buildFeedEntries()` method and optionally override the `buildFeedMetadata()` method (the default implementation is empty), as shown below.

```
public class SampleContentAtomView extends AbstractAtomFeedView {

    @Override
    protected void buildFeedMetadata(Map<String, Object> model,
            Feed feed, HttpServletRequest request) {
        // implementation omitted
    }

    @Override
    protected List<Entry> buildFeedEntries(Map<String, Object> model,
            HttpServletRequest request, HttpServletResponse response) throws Exception
{
        // implementation omitted
    }

}
```

Similar requirements apply for implementing `AbstractRssFeedView`, as shown below.

```
public class SampleContentAtomView extends AbstractRssFeedView {

    @Override
    protected void buildFeedMetadata(Map<String, Object> model,
            Channel feed, HttpServletRequest request) {
        // implementation omitted
    }

    @Override
    protected List<Item> buildFeedItems(Map<String, Object> model,
            HttpServletRequest request, HttpServletResponse response) throws Exception
{
        // implementation omitted
    }

}
```

The `buildFeedItems()` and `buildFeedEntires()` methods pass in the HTTP request in case you need to access the Locale. The HTTP response is passed in only for the setting of cookies or other HTTP headers. The feed will automatically be written to the response object after the method returns.

For an example of creating an Atom view please refer to Alef Arendsen's Spring Team Blog entry.

2.12. JSON Mapping View

The `MappingJackson2JsonView` uses the Jackson library's `ObjectMapper` to render the response content as JSON. By default, the entire contents of the model map (with the exception of framework-specific classes) will be encoded as JSON. For cases where the contents of the map need to be filtered, users may specify a specific set of model attributes to encode via the `RenderedAttributes` property. The `extractValueFromSingleKeyModel` property may also be used to have the value in single-key models extracted and serialized directly rather than as a map of model attributes.

JSON mapping can be customized as needed through the use of Jackson's provided annotations. When further control is needed, a custom `ObjectMapper` can be injected through the `ObjectMapper` property for cases where custom JSON serializers/deserializers need to be provided for specific types.

JSONP is supported and automatically enabled when the request has a query parameter named `jsonp` or `callback`. The JSONP query parameter name(s) could be customized through the `jsonpParameterNames` property.

2.13. XML Mapping View

The `MappingJackson2XmlView` uses the Jackson XML extension's `XmlMapper` to render the response content as XML. If the model contains multiples entries, the object to be serialized should be set explicitly using the `modelKey` bean property. If the model contains a single entry, it will be serialized automatically.

XML mapping can be customized as needed through the use of JAXB or Jackson's provided annotations. When further control is needed, a custom `XmlMapper` can be injected through the `ObjectMapper` property for cases where custom XML serializers/deserializers need to be provided for specific types.

Chapter 3. Spring MVC CORS Support

3.1. Introduction

For security reasons, browsers prohibit AJAX calls to resources residing outside the current origin. For example, as you're checking your bank account in one tab, you could have the evil.com website open in another tab. The scripts from evil.com should not be able to make AJAX requests to your bank API (e.g., withdrawing money from your account!) using your credentials.

Cross-origin resource sharing (CORS) is a W3C specification implemented by most browsers that allows you to specify in a flexible way what kind of cross domain requests are authorized, instead of using some less secured and less powerful hacks like IFRAME or JSONP.

As of Spring Framework 4.2, CORS is supported out of the box. CORS requests (including preflight ones with an `OPTIONS` method) are automatically dispatched to the various registered `HandlerMapping`s. They handle CORS preflight requests and intercept CORS simple and actual requests thanks to a CorsProcessor implementation (DefaultCorsProcessor by default) in order to add the relevant CORS response headers (like `Access-Control-Allow-Origin`) based on the CORS configuration you have provided.

Since CORS requests are automatically dispatched, you **do not need** to change the `DispatcherServlet dispatchOptionsRequest` init parameter value; using its default value (`false`) is the recommended approach.

3.2. Controller method CORS configuration

You can add an `@CrossOrigin` annotation to your `@RequestMapping` annotated handler method in order to enable CORS on it. By default `@CrossOrigin` allows all origins and the HTTP methods specified in the `@RequestMapping` annotation:

```
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin
    @RequestMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @RequestMapping(method = RequestMethod.DELETE, path = "/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

It is also possible to enable CORS for the whole controller:

```
@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @RequestMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @RequestMapping(method = RequestMethod.DELETE, path = "/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

In the above example CORS support is enabled for both the `retrieve()` and the `remove()` handler methods, and you can also see how you can customize the CORS configuration using `@CrossOrigin` attributes.

You can even use both controller-level and method-level CORS configurations; Spring will then combine attributes from both annotations to create merged CORS configuration.

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("http://domain2.com")
    @RequestMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @RequestMapping(method = RequestMethod.DELETE, path = "/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

3.3. Global CORS configuration

In addition to fine-grained, annotation-based configuration you'll probably want to define some global CORS configuration as well. This is similar to using filters but can be declared within Spring MVC and combined with fine-grained `@CrossOrigin` configuration. By default all origins and `GET`, `HEAD`, and `POST` methods are allowed.

3.3.1. JavaConfig

Enabling CORS for the whole application is as simple as:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**");
    }
}
```

You can easily change any properties, as well as only apply this CORS configuration to a specific path pattern:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins("http://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(false).maxAge(3600);
    }
}
```

3.3.2. XML namespace

The following minimal XML configuration enables CORS for the `/**` path pattern with the same default properties as with the aforementioned JavaConfig examples:

```
<mvc:cors>
    <mvc:mapping path="/**" />
</mvc:cors>
```

It is also possible to declare several CORS mappings with customized properties:

```
<mvc:cors>

    <mvc:mapping path="/api/**"
        allowed-origins="http://domain1.com, http://domain2.com"
        allowed-methods="GET, PUT"
        allowed-headers="header1, header2, header3"
        exposed-headers="header1, header2" allow-credentials="false"
        max-age="123" />

    <mvc:mapping path="/resources/**"
        allowed-origins="http://domain1.com" />

</mvc:cors>
```

3.4. Advanced Customization

CorsConfiguration allows you to specify how the CORS requests should be processed: allowed origins, headers, methods, etc. It can be provided in various ways:

- `AbstractHandlerMapping#setCorsConfiguration()` allows to specify a `Map` with several CorsConfiguration instances mapped to path patterns like `/api/**`.
- Subclasses can provide their own `CorsConfiguration` by overriding the `AbstractHandlerMapping#getCorsConfiguration(Object, HttpServletRequest)` method.
- Handlers can implement the `CorsConfigurationSource` interface (like `ResourceHttpRequestHandler` now does) in order to provide a CorsConfiguration instance for each request.

3.5. Filter based CORS support

In order to support CORS with filter-based security frameworks like Spring Security, or with other libraries that do not support natively CORS, Spring Framework also provides a `CorsFilter`. Instead of using `@CrossOrigin` or `WebMvcConfigurer#addCorsMappings(CorsRegistry)`, you need to register a custom filter defined like bellow:

```
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.filter.CorsFilter;

public class MyCorsFilter extends CorsFilter {

    public MyCorsFilter() {
        super(configurationSource());
    }

    private static UrlBasedCorsConfigurationSource configurationSource() {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("http://domain1.com");
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource(
);
        source.registerCorsConfiguration("/**", config);
        return source;
    }
}
```

You need to ensure that `CorsFilter` is ordered before the other filters, see this blog post about how to configure Spring Boot accordingly.

Chapter 4. Servlet Stack WebSocket Support

This part of the reference documentation covers Spring Framework's support for WebSocket-style messaging in web applications including use of STOMP as an application level WebSocket sub-protocol.

Introduction establishes a frame of mind in which to think about WebSocket, covering adoption challenges, design considerations, and thoughts on when it is a good fit.

WebSocket API reviews the Spring WebSocket API on the server-side, while SockJS Fallback Options explains the SockJS protocol and shows how to configure and use it.

Overview of STOMP introduces the STOMP messaging protocol. Enable STOMP over WebSocket demonstrates how to configure STOMP support in Spring. Annotation Message Handling and the following sections explain how to write annotated message handling methods, send messages, choose message broker options, as well as work with the special "user" destinations. Finally, Testing Annotated Controller Methods lists three approaches to testing STOMP/WebSocket applications.

4.1. Introduction

The WebSocket protocol RFC 6455 defines an important new capability for web applications: full-duplex, two-way communication between client and server. It is an exciting new capability on the heels of a long history of techniques to make the web more interactive including Java Applets, XMLHttpRequest, Adobe Flash, ActiveXObject, various Comet techniques, server-sent events, and others.

A proper introduction to the WebSocket protocol is beyond the scope of this document. At a minimum however it's important to understand that HTTP is used only for the initial handshake, which relies on a mechanism built into HTTP to request a protocol upgrade (or in this case a protocol switch) to which the server can respond with HTTP status 101 (switching protocols) if it agrees. Assuming the handshake succeeds the TCP socket underlying the HTTP upgrade request remains open and both client and server can use it to send messages to each other.

Spring Framework 4 includes a new `spring-websocket` module with comprehensive WebSocket support. It is compatible with the Java WebSocket API standard (JSR-356) and also provides additional value-add as explained in the rest of the introduction.

4.1.1. WebSocket Fallback Options

An important challenge to adoption is the lack of support for WebSocket in some browsers. Notably the first Internet Explorer version to support WebSocket is version 10 (see http://caniuse.com/websockets for support by browser versions). Furthermore, some restrictive proxies may be configured in ways that either preclude the attempt to do an HTTP upgrade or otherwise break connection after some time because it has remained opened for too long. A good overview on this topic from Peter Lubbers is available in the InfoQ article "How HTML5 Web Sockets Interact With Proxy Servers".

Therefore to build a WebSocket application today, fallback options are required in order to simulate

the WebSocket API where necessary. The Spring Framework provides such transparent fallback options based on the SockJS protocol. These options can be enabled through configuration and do not require modifying the application otherwise.

4.1.2. A Messaging Architecture

Aside from short-to-midterm adoption challenges, using WebSocket brings up important design considerations that are important to recognize early on, especially in contrast to what we know about building web applications today.

Today REST is a widely accepted, understood, and supported architecture for building web applications. It is an architecture that relies on having many URLs (*nouns*), a handful of HTTP methods (*verbs*), and other principles such as using hypermedia (*links*), remaining stateless, etc.

By contrast a WebSocket application may use a single URL only for the initial HTTP handshake. All messages thereafter share and flow on the same TCP connection. This points to an entirely different, asynchronous, event-driven, messaging architecture. One that is much closer to traditional messaging applications (e.g. JMS, AMQP).

Spring Framework 4 includes a new `spring-messaging` module with key abstractions from the Spring Integration project such as `Message`, `MessageChannel`, `MessageHandler`, and others that can serve as a foundation for such a messaging architecture. The module also includes a set of annotations for mapping messages to methods, similar to the Spring MVC annotation based programming model.

4.1.3. Sub-Protocol Support in WebSocket

WebSocket does imply a *messaging architecture* but does not mandate the use of any specific *messaging protocol*. It is a very thin layer over TCP that transforms a stream of bytes into a stream of messages (either text or binary) and not much more. It is up to applications to interpret the meaning of a message.

Unlike HTTP, which is an application-level protocol, in the WebSocket protocol there is simply not enough information in an incoming message for a framework or container to know how to route it or process it. Therefore WebSocket is arguably too low level for anything but a very trivial application. It can be done, but it will likely lead to creating a framework on top. This is comparable to how most web applications today are written using a web framework rather than the Servlet API alone.

For this reason the WebSocket RFC defines the use of sub-protocols. During the handshake, the client and server can use the header `Sec-WebSocket-Protocol` to agree on a sub-protocol, i.e. a higher, application-level protocol to use. The use of a sub-protocol is not required, but even if not used, applications will still need to choose a message format that both the client and server can understand. That format can be custom, framework-specific, or a standard messaging protocol.

The Spring Framework provides support for using STOMP — a simple, messaging protocol originally created for use in scripting languages with frames inspired by HTTP. STOMP is widely supported and well suited for use over WebSocket and over the web.

4.1.4. Should I Use WebSocket?

With all the design considerations surrounding the use of WebSocket, it is reasonable to ask, "When is it appropriate to use?".

The best fit for WebSocket is in web applications where the client and server need to exchange events at high frequency and with low latency. Prime candidates include, but are not limited to, applications in finance, games, collaboration, and others. Such applications are both very sensitive to time delays and also need to exchange a wide variety of messages at a high frequency.

For other application types, however, this may not be the case. For example, a news or social feed that shows breaking news as it becomes available may be perfectly okay with simple polling once every few minutes. Here latency is important, but it is acceptable if the news takes a few minutes to appear.

Even in cases where latency is crucial, if the volume of messages is relatively low (e.g. monitoring network failures) the use of long polling should be considered as a relatively simple alternative that works reliably and is comparable in terms of efficiency (again assuming the volume of messages is relatively low).

It is the combination of both low latency and high frequency of messages that can make the use of the WebSocket protocol critical. Even in such applications, the choice remains whether all client-server communication should be done through WebSocket messages as opposed to using HTTP and REST. The answer is going to vary by application; however, it is likely that some functionality may be exposed over both WebSocket and as a REST API in order to provide clients with alternatives. Furthermore, a REST API call may need to broadcast a message to interested clients connected via WebSocket.

The Spring Framework allows `@Controller` and `@RestController` classes to have both HTTP request handling and WebSocket message handling methods. Furthermore, a Spring MVC request handling method, or any application method for that matter, can easily broadcast a message to all interested WebSocket clients or to a specific user.

4.2. WebSocket API

The Spring Framework provides a WebSocket API designed to adapt to various WebSocket engines. Currently the list includes WebSocket runtimes such as Tomcat 7.0.47+, Jetty 9.1+, GlassFish 4.1+, WebLogic 12.1.3+, and Undertow 1.0+ (and WildFly 8.0+). Additional support may be added as more WebSocket runtimes become available.

As explained in the introduction, direct use of a WebSocket API is too low level for applications — until assumptions are made about the format of a message there is little a framework can do to interpret messages or route them via annotations. This is why applications should consider using a sub-protocol and Spring's STOMP over WebSocket support.

When using a higher level protocol, the details of the WebSocket API become less relevant, much like the details of TCP communication are not exposed to applications when using HTTP. Nevertheless this section covers the details of using WebSocket directly.

4.2.1. Create and Configure a WebSocketHandler

Creating a WebSocket server is as simple as implementing `WebSocketHandler` or more likely extending either `TextWebSocketHandler` or `BinaryWebSocketHandler`:

```
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.TextMessage;

public class MyHandler extends TextWebSocketHandler {

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) {
        // ...
    }

}
```

There is dedicated WebSocket Java-config and XML namespace support for mapping the above WebSocket handler to a specific URL:

```
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

XML configuration equivalent:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

The above is for use in Spring MVC applications and should be included in the configuration of a DispatcherServlet. However, Spring's WebSocket support does not depend on Spring MVC. It is relatively simple to integrate a `WebSocketHandler` into other HTTP serving environments with the help of WebSocketHttpRequestHandler.

4.2.2. Customizing the WebSocket Handshake

The easiest way to customize the initial HTTP WebSocket handshake request is through a `HandshakeInterceptor`, which exposes "before" and "after" the handshake methods. Such an

interceptor can be used to preclude the handshake or to make any attributes available to the `WebSocketSession`. For example, there is a built-in interceptor for passing HTTP session attributes to the WebSocket session:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(new MyHandler(), "/myHandler")
            .addInterceptors(new HttpSessionHandshakeInterceptor());
    }

}
```

And the XML configuration equivalent:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
        <websocket:handshake-interceptors>
            <bean class=
"org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor"/>
        </websocket:handshake-interceptors>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

A more advanced option is to extend the `DefaultHandshakeHandler` that performs the steps of the WebSocket handshake, including validating the client origin, negotiating a sub-protocol, and others. An application may also need to use this option if it needs to configure a custom `RequestUpgradeStrategy` in order to adapt to a WebSocket server engine and version that is not yet supported (also see Deployment Considerations for more on this subject). Both the Java-config and XML namespace make it possible to configure a custom `HandshakeHandler`.

4.2.3. WebSocketHandler Decoration

Spring provides a `WebSocketHandlerDecorator` base class that can be used to decorate a `WebSocketHandler` with additional behavior. Logging and exception handling implementations are provided and added by default when using the WebSocket Java-config or XML namespace. The `ExceptionWebSocketHandlerDecorator` catches all uncaught exceptions arising from any WebSocketHandler method and closes the WebSocket session with status `1011` that indicates a server error.

4.2.4. Deployment Considerations

The Spring WebSocket API is easy to integrate into a Spring MVC application where the `DispatcherServlet` serves both HTTP WebSocket handshake as well as other HTTP requests. It is also easy to integrate into other HTTP processing scenarios by invoking `WebSocketHttpRequestHandler`. This is convenient and easy to understand. However, special considerations apply with regards to JSR-356 runtimes.

The Java WebSocket API (JSR-356) provides two deployment mechanisms. The first involves a Servlet container classpath scan (Servlet 3 feature) at startup; and the other is a registration API to use at Servlet container initialization. Neither of these mechanism makes it possible to use a single "front controller" for all HTTP processing — including WebSocket handshake and all other HTTP requests — such as Spring MVC's `DispatcherServlet`.

This is a significant limitation of JSR-356 that Spring's WebSocket support addresses by providing a server-specific `RequestUpgradeStrategy` even when running in a JSR-356 runtime.

A request to overcome the above limitation in the Java WebSocket API has been created and can be followed at WEBSOCKET_SPEC-211. Also note that Tomcat and Jetty already provide native API alternatives that makes it easy to overcome the limitation. We are hopeful that more servers will follow their example regardless of when it is addressed in the Java WebSocket API.

A secondary consideration is that Servlet containers with JSR-356 support are expected to perform a `ServletContainerInitializer` (SCI) scan that can slow down application startup, in some cases dramatically. If a significant impact is observed after an upgrade to a Servlet container version with JSR-356 support, it should be possible to selectively enable or disable web fragments (and SCI scanning) through the use of the `<absolute-ordering />` element in `web.xml`:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <absolute-ordering/>

</web-app>
```

You can then selectively enable web fragments by name, such as Spring's own `SpringServletContainerInitializer` that provides support for the Servlet 3 Java initialization API, if required:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <absolute-ordering>
        <name>spring_web</name>
    </absolute-ordering>

</web-app>
```

4.2.5. Configuring the WebSocket Engine

Each underlying WebSocket engine exposes configuration properties that control runtime characteristics such as the size of message buffer sizes, idle timeout, and others.

For Tomcat, WildFly, and GlassFish add a `ServletServerContainerFactoryBean` to your WebSocket Java config:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Bean
    public ServletServerContainerFactoryBean createWebSocketContainer() {
        ServletServerContainerFactoryBean container = new
ServletServerContainerFactoryBean();
        container.setMaxTextMessageBufferSize(8192);
        container.setMaxBinaryMessageBufferSize(8192);
        return container;
    }

}
```

or WebSocket XML namespace:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <bean class="org.springframework...ServletServerContainerFactoryBean">
        <property name="maxTextMessageBufferSize" value="8192"/>
        <property name="maxBinaryMessageBufferSize" value="8192"/>
    </bean>

</beans>
```

For client side WebSocket configuration, you should use `WebSocketContainerFactoryBean` (XML) or `ContainerProvider.getWebSocketContainer()` (Java config).

For Jetty, you'll need to supply a pre-configured Jetty `WebSocketServerFactory` and plug that into Spring's `DefaultHandshakeHandler` through your WebSocket Java config:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(echoWebSocketHandler(),
            "/echo").setHandshakeHandler(handshakeHandler());
    }

    @Bean
    public DefaultHandshakeHandler handshakeHandler() {

        WebSocketPolicy policy = new WebSocketPolicy(WebSocketBehavior.SERVER);
        policy.setInputBufferSize(8192);
        policy.setIdleTimeout(600000);

        return new DefaultHandshakeHandler(
                new JettyRequestUpgradeStrategy(new WebSocketServerFactory(policy)));
    }

}
```

or WebSocket XML namespace:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/echo" handler="echoHandler"/>
        <websocket:handshake-handler ref="handshakeHandler"/>
    </websocket:handlers>

    <bean id="handshakeHandler" class="org.springframework...DefaultHandshakeHandler">
        <constructor-arg ref="upgradeStrategy"/>
    </bean>

    <bean id="upgradeStrategy" class=
"org.springframework...JettyRequestUpgradeStrategy">
        <constructor-arg ref="serverFactory"/>
    </bean>

    <bean id="serverFactory" class="org.eclipse.jetty...WebSocketServerFactory">
        <constructor-arg>
            <bean class="org.eclipse.jetty...WebSocketPolicy">
                <constructor-arg value="SERVER"/>
                <property name="inputBufferSize" value="8092"/>
                <property name="idleTimeout" value="600000"/>
            </bean>
        </constructor-arg>
    </bean>

</beans>
```

4.2.6. Configuring allowed origins

As of Spring Framework 4.1.5, the default behavior for WebSocket and SockJS is to accept only *same origin* requests. It is also possible to allow *all* or a specified list of origins. This check is mostly designed for browser clients. There is nothing preventing other types of clients from modifying the `Origin` header value (see RFC 6454: The Web Origin Concept for more details).

The 3 possible behaviors are:

- Allow only same origin requests (default): in this mode, when SockJS is enabled, the Iframe HTTP response header `X-Frame-Options` is set to `SAMEORIGIN`, and JSONP transport is disabled since it does not allow to check the origin of a request. As a consequence, IE6 and IE7 are not supported when this mode is enabled.
- Allow a specified list of origins: each provided *allowed origin* must start with `http://` or

https://. In this mode, when SockJS is enabled, both IFrame and JSONP based transports are disabled. As a consequence, IE6 through IE9 are not supported when this mode is enabled.

- Allow all origins: to enable this mode, you should provide * as the allowed origin value. In this mode, all transports are available.

WebSocket and SockJS allowed origins can be configured as shown bellow:

```
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler").setAllowedOrigins(
"http://mydomain.com");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

XML configuration equivalent:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers allowed-origins="http://mydomain.com">
        <websocket:mapping path="/myHandler" handler="myHandler" />
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

4.3. SockJS Fallback Options

As explained in the introduction, WebSocket is not supported in all browsers yet and may be precluded by restrictive network proxies. This is why Spring provides fallback options that emulate the WebSocket API as close as possible based on the SockJS protocol (version 0.3.3).

4.3.1. Overview of SockJS

The goal of SockJS is to let applications use a WebSocket API but fall back to non-WebSocket alternatives when necessary at runtime, i.e. without the need to change application code.

SockJS consists of:

- The SockJS protocol defined in the form of executable narrated tests.
- The SockJS JavaScript client - a client library for use in browsers.
- SockJS server implementations including one in the Spring Framework `spring-websocket` module.
- As of 4.1 `spring-websocket` also provides a SockJS Java client.

SockJS is designed for use in browsers. It goes to great lengths to support a wide range of browser versions using a variety of techniques. For the full list of SockJS transport types and browsers see the SockJS client page. Transports fall in 3 general categories: WebSocket, HTTP Streaming, and HTTP Long Polling. For an overview of these categories see this blog post.

The SockJS client begins by sending `"GET /info"` to obtain basic information from the server. After that it must decide what transport to use. If possible WebSocket is used. If not, in most browsers there is at least one HTTP streaming option and if not then HTTP (long) polling is used.

All transport requests have the following URL structure:

```
http://host:port/myApp/myEndpoint/{server-id}/{session-id}/{transport}
```

- `{server-id}` - useful for routing requests in a cluster but not used otherwise.
- `{session-id}` - correlates HTTP requests belonging to a SockJS session.
- `{transport}` - indicates the transport type, e.g. "websocket", "xhr-streaming", etc.

The WebSocket transport needs only a single HTTP request to do the WebSocket handshake. All messages thereafter are exchanged on that socket.

HTTP transports require more requests. Ajax/XHR streaming for example relies on one long-running request for server-to-client messages and additional HTTP POST requests for client-to-server messages. Long polling is similar except it ends the current request after each server-to-client send.

SockJS adds minimal message framing. For example the server sends the letter o ("open" frame) initially, messages are sent as a["message1","message2"] (JSON-encoded array), the letter h ("heartbeat" frame) if no messages flow for 25 seconds by default, and the letter c ("close" frame) to

close the session.

To learn more, run an example in a browser and watch the HTTP requests. The SockJS client allows fixing the list of transports so it is possible to see each transport one at a time. The SockJS client also provides a debug flag which enables helpful messages in the browser console. On the server side enable `TRACE` logging for `org.springframework.web.socket`. For even more detail refer to the SockJS protocol narrated test.

4.3.2. Enable SockJS

SockJS is easy to enable through Java configuration:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler").withSockJS();
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

and the XML configuration equivalent:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
        <websocket:sockjs/>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

The above is for use in Spring MVC applications and should be included in the configuration of a DispatcherServlet. However, Spring's WebSocket and SockJS support does not depend on Spring MVC. It is relatively simple to integrate into other HTTP serving environments with the help of SockJsHttpRequestHandler.

On the browser side, applications can use the sockjs-client (version 1.0.x) that emulates the W3C WebSocket API and communicates with the server to select the best transport option depending on the browser it's running in. Review the sockjs-client page and the list of transport types supported by browser. The client also provides several configuration options, for example, to specify which transports to include.

4.3.3. HTTP Streaming in IE 8, 9: Ajax/XHR vs IFrame

Internet Explorer 8 and 9 are and will remain common for some time. They are a key reason for having SockJS. This section covers important considerations about running in those browsers.

The SockJS client supports Ajax/XHR streaming in IE 8 and 9 via Microsoft's XDomainRequest. That works across domains but does not support sending cookies. Cookies are very often essential for Java applications. However since the SockJS client can be used with many server types (not just Java ones), it needs to know whether cookies matter. If so the SockJS client prefers Ajax/XHR for streaming or otherwise it relies on a iframe-based technique.

The very first `"/info"` request from the SockJS client is a request for information that can influence the client's choice of transports. One of those details is whether the server application relies on cookies, e.g. for authentication purposes or clustering with sticky sessions. Spring's SockJS support includes a property called `sessionCookieNeeded`. It is enabled by default since most Java applications rely on the `JSESSIONID` cookie. If your application does not need it, you can turn off this option and the SockJS client should choose `xdr-streaming` in IE 8 and 9.

If you do use an iframe-based transport, and in any case, it is good to know that browsers can be instructed to block the use of IFrames on a given page by setting the HTTP response header `X-Frame-Options` to `DENY`, `SAMEORIGIN`, or `ALLOW-FROM <origin>`. This is used to prevent clickjacking.

Spring Security 3.2+ provides support for setting `X-Frame-Options` on every response. By default the Spring Security Java config sets it to `DENY`. In 3.2 the Spring Security XML namespace does not set that header by default but may be configured to do so, and in the future it may set it by default.

See Section 7.1. "Default Security Headers" of the Spring Security documentation for details on how to configure the setting of the `X-Frame-Options` header. You may also check or watch SEC-2501 for additional background.

If your application adds the `X-Frame-Options` response header (as it should!) and relies on an iframe-based transport, you will need to set the header value to `SAMEORIGIN` or `ALLOW-FROM <origin>`. Along with that the Spring SockJS support also needs to know the location of the SockJS client because it is loaded from the iframe. By default the iframe is set to download the SockJS client from a CDN location. It is a good idea to configure this option to a URL from the same origin as the application.

In Java config this can be done as shown below. The XML namespace provides a similar option via

the `<websocket:sockjs>` element:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS()
                .setClientLibraryUrl("http://localhost:8080/myapp/js/sockjs-client.js
");
    }

    // ...

}
```

During initial development, do enable the SockJS client `devel` mode that prevents the browser from caching SockJS requests (like the iframe) that would otherwise be cached. For details on how to enable it see the SockJS client page.

4.3.4. Heartbeat Messages

The SockJS protocol requires servers to send heartbeat messages to preclude proxies from concluding a connection is hung. The Spring SockJS configuration has a property called `heartbeatTime` that can be used to customize the frequency. By default a heartbeat is sent after 25 seconds assuming no other messages were sent on that connection. This 25 seconds value is in line with the following IETF recommendation for public Internet applications.

When using STOMP over WebSocket/SockJS, if the STOMP client and server negotiate heartbeats to be exchanged, the SockJS heartbeats are disabled.

The Spring SockJS support also allows configuring the `TaskScheduler` to use for scheduling heartbeats tasks. The task scheduler is backed by a thread pool with default settings based on the number of available processors. Applications should consider customizing the settings according to their specific needs.

4.3.5. Servlet 3 Async Requests

HTTP streaming and HTTP long polling SockJS transports require a connection to remain open longer than usual. For an overview of these techniques see this blog post.

In Servlet containers this is done through Servlet 3 async support that allows exiting the Servlet container thread processing a request and continuing to write to the response from another thread.

A specific issue is that the Servlet API does not provide notifications for a client that has gone away, see SERVLET_SPEC-44. However, Servlet containers raise an exception on subsequent attempts to write to the response. Since Spring's SockJS Service supports sever-sent heartbeats (every 25

seconds by default), that means a client disconnect is usually detected within that time period or earlier if messages are sent more frequently.

As a result network IO failures may occur simply because a client has disconnected, which can fill the log with unnecessary stack traces. Spring makes a best effort to identify such network failures that represent client disconnects (specific to each server) and log a minimal message using the dedicated log category `DISCONNECTED_CLIENT_LOG_CATEGORY` defined in `AbstractSockJsSession`. If you need to see the stack traces, set that log category to TRACE.

4.3.6. CORS Headers for SockJS

If you allow cross-origin requests (see Configuring allowed origins), the SockJS protocol uses CORS for cross-domain support in the XHR streaming and polling transports. Therefore CORS headers are added automatically unless the presence of CORS headers in the response is detected. So if an application is already configured to provide CORS support, e.g. through a Servlet Filter, Spring's SockJsService will skip this part.

It is also possible to disable the addition of these CORS headers via the `suppressCors` property in Spring's SockJsService.

The following is the list of headers and values expected by SockJS:

- `"Access-Control-Allow-Origin"` - initialized from the value of the "Origin" request header.
- `"Access-Control-Allow-Credentials"` - always set to `true`.
- `"Access-Control-Request-Headers"` - initialized from values from the equivalent request header.
- `"Access-Control-Allow-Methods"` - the HTTP methods a transport supports (see `TransportType` enum).
- `"Access-Control-Max-Age"` - set to 31536000 (1 year).

For the exact implementation see `addCorsHeaders` in `AbstractSockJsService` as well as the `TransportType` enum in the source code.

Alternatively if the CORS configuration allows it consider excluding URLs with the SockJS endpoint prefix thus letting Spring's `SockJsService` handle it.

4.3.7. SockJS Client

A SockJS Java client is provided in order to connect to remote SockJS endpoints without using a browser. This can be especially useful when there is a need for bidirectional communication between 2 servers over a public network, i.e. where network proxies may preclude the use of the WebSocket protocol. A SockJS Java client is also very useful for testing purposes, for example to simulate a large number of concurrent users.

The SockJS Java client supports the "websocket", "xhr-streaming", and "xhr-polling" transports. The remaining ones only make sense for use in a browser.

The `WebSocketTransport` can be configured with:

- `StandardWebSocketClient` in a JSR-356 runtime
- `JettyWebSocketClient` using the Jetty 9+ native WebSocket API
- Any implementation of Spring's `WebSocketClient`

An `XhrTransport` by definition supports both "xhr-streaming" and "xhr-polling" since from a client perspective there is no difference other than in the URL used to connect to the server. At present there are two implementations:

- `RestTemplateXhrTransport` uses Spring's `RestTemplate` for HTTP requests.
- `JettyXhrTransport` uses Jetty's `HttpClient` for HTTP requests.

The example below shows how to create a SockJS client and connect to a SockJS endpoint:

```
List<Transport> transports = new ArrayList<>(2);
transports.add(new WebSocketTransport(new StandardWebSocketClient()));
transports.add(new RestTemplateXhrTransport());

SockJsClient sockJsClient = new SockJsClient(transports);
sockJsClient.doHandshake(new MyWebSocketHandler(), "ws://example.com:8080/sockjs");
```

SockJS uses JSON formatted arrays for messages. By default Jackson 2 is used and needs to be on the classpath. Alternatively you can configure a custom implementation of `SockJsMessageCodec` and configure it on the `SockJsClient`.

To use the SockJsClient for simulating a large number of concurrent users you will need to configure the underlying HTTP client (for XHR transports) to allow a sufficient number of connections and threads. For example with Jetty:

```
HttpClient jettyHttpClient = new HttpClient();
jettyHttpClient.setMaxConnectionsPerDestination(1000);
jettyHttpClient.setExecutor(new QueuedThreadPool(1000));
```

Consider also customizing these server-side SockJS related properties (see Javadoc for details):

```
@Configuration
public class WebSocketConfig extends WebSocketMessageBrokerConfigurationSupport {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/sockjs").withSockJS()
            .setStreamBytesLimit(512 * 1024)
            .setHttpMessageCacheSize(1000)
            .setDisconnectDelay(30 * 1000);
    }

    // ...

}
```

4.4. STOMP Over WebSocket Messaging Architecture

The WebSocket protocol defines two types of messages, text and binary, but their content is undefined. It's expected that the client and server may agree on using a sub-protocol (i.e. a higher-level protocol) to define message semantics. While the use of a sub-protocol with WebSocket is completely optional either way client and server will need to agree on some kind of protocol to help interpret messages.

4.4.1. Overview of STOMP

STOMP is a simple text-oriented messaging protocol that was originally created for scripting languages such as Ruby, Python, and Perl to connect to enterprise message brokers. It is designed to address a subset of commonly used messaging patterns. STOMP can be used over any reliable 2-way streaming network protocol such as TCP and WebSocket. Although STOMP is a text-oriented protocol, the payload of messages can be either text or binary.

STOMP is a frame based protocol whose frames are modeled on HTTP. The structure of a STOMP frame:

```
COMMAND
header1:value1
header2:value2

Body^@
```

Clients can use the SEND or SUBSCRIBE commands to send or subscribe for messages along with a "destination" header that describes what the message is about and who should receive it. This enables a simple publish-subscribe mechanism that can be used to send messages through the broker to other connected clients or to send messages to the server to request that some work be performed.

When using Spring's STOMP support, the Spring WebSocket application acts as the STOMP broker to

clients. Messages are routed to `@Controller` message-handling methods or to a simple, in-memory broker that keeps track of subscriptions and broadcasts messages to subscribed users. You can also configure Spring to work with a dedicated STOMP broker (e.g. RabbitMQ, ActiveMQ, etc) for the actual broadcasting of messages. In that case Spring maintains TCP connections to the broker, relays messages to it, and also passes messages from it down to connected WebSocket clients. Thus Spring web applications can rely on unified HTTP-based security, common validation, and a familiar programming model message-handling work.

Here is an example of a client subscribing to receive stock quotes which the server may emit periodically e.g. via a scheduled task sending messages through a `SimpMessagingTemplate` to the broker:

```
SUBSCRIBE
id:sub-1
destination:/topic/price.stock.*

^@
```

Here is an example of a client sending a trade request, which the server may handle through an `@MessageMapping` method and later on, after the execution, broadcast a trade confirmation message and details down to the client:

```
SEND
destination:/queue/trade
content-type:application/json
content-length:44

{"action":"BUY","ticker":"MMM","shares",44}^@
```

The meaning of a destination is intentionally left opaque in the STOMP spec. It can be any string, and it's entirely up to STOMP servers to define the semantics and the syntax of the destinations that they support. It is very common, however, for destinations to be path-like strings where `"/topic/.."` implies publish-subscribe (*one-to-many*) and `"/queue/"` implies point-to-point (*one-to-one*) message exchanges.

STOMP servers can use the MESSAGE command to broadcast messages to all subscribers. Here is an example of a server sending a stock quote to a subscribed client:

```
MESSAGE
message-id:nxahklf6-1
subscription:sub-1
destination:/topic/price.stock.MMM

{"ticker":"MMM","price":129.45}^@
```

It is important to know that a server cannot send unsolicited messages. All messages from a server

must be in response to a specific client subscription, and the "subscription-id" header of the server message must match the "id" header of the client subscription.

The above overview is intended to provide the most basic understanding of the STOMP protocol. It is recommended to review the protocol specification in full.

The benefits of using STOMP as a WebSocket sub-protocol:

- No need to invent a custom message format
- Use existing stomp.js client in the browser
- Ability to route messages to based on destination
- Option to use full-fledged message broker such as RabbitMQ, ActiveMQ, etc. for broadcasting

Most importantly the use of STOMP (vs plain WebSocket) enables the Spring Framework to provide a programming model for application-level use in the same way that Spring MVC provides a programming model based on HTTP.

4.4.2. Enable STOMP over WebSocket

The Spring Framework provides support for using STOMP over WebSocket through the spring-messaging and spring-websocket modules. Here is an example of exposing a STOMP WebSocket/SockJS endpoint at the URL path `/portfolio` where messages whose destination starts with "/app" are routed to message-handling methods (i.e. application work) and messages whose destinations start with "/topic" or "/queue" will be routed to the message broker (i.e. broadcasting to other connected clients):

```
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS();
    }

    @Override
       public void configureMessageBroker(MessageBrokerRegistry config) {
           config.setApplicationDestinationPrefixes("/app");
           config.enableSimpleBroker("/topic", "/queue");
       }

}
```

and in XML:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker application-destination-prefix="/app">
        <websocket:stomp-endpoint path="/portfolio">
            <websocket:sockjs/>
        </websocket:stomp-endpoint>
        <websocket:simple-broker prefix="/topic, /queue"/>
    </websocket:message-broker>

</beans>
```

The "/app" prefix is arbitrary. You can pick any prefix. It's simply meant to differentiate messages to be routed to message-handling methods to do application work vs messages to be routed to the broker to broadcast to subscribed clients.

The "/topic" and "/queue" prefixes depend on the broker in use. In the case of the simple, in-memory broker the prefixes do not have any special meaning; it's merely a convention that indicates how the destination is used (pub-sub targetting many subscribers or point-to-point messages typically targeting an individual recipient). In the case of using a dedicated broker, most brokers use "/topic" as a prefix for destinations with pub-sub semantics and "/queue" for destinations with point-to-point semantics. Check the STOMP page of the broker to see the destination semantics it supports.

On the browser side, a client might connect as follows using stomp.js and the sockjs-client:

```
var socket = new SockJS("/spring-websocket-portfolio/portfolio");
var stompClient = Stomp.over(socket);

stompClient.connect({}, function(frame) {
}
```

Or if connecting via WebSocket (without SockJS):

```
var socket = new WebSocket("/spring-websocket-portfolio/portfolio");
var stompClient = Stomp.over(socket);

stompClient.connect({}, function(frame) {
}
```

Note that the `stompClient` above does not need to specify `login` and `passcode` headers. Even if it did, they would be ignored, or rather overridden, on the server side. See the sections Connections To Full-Featured Broker and Authentication for more information on authentication.

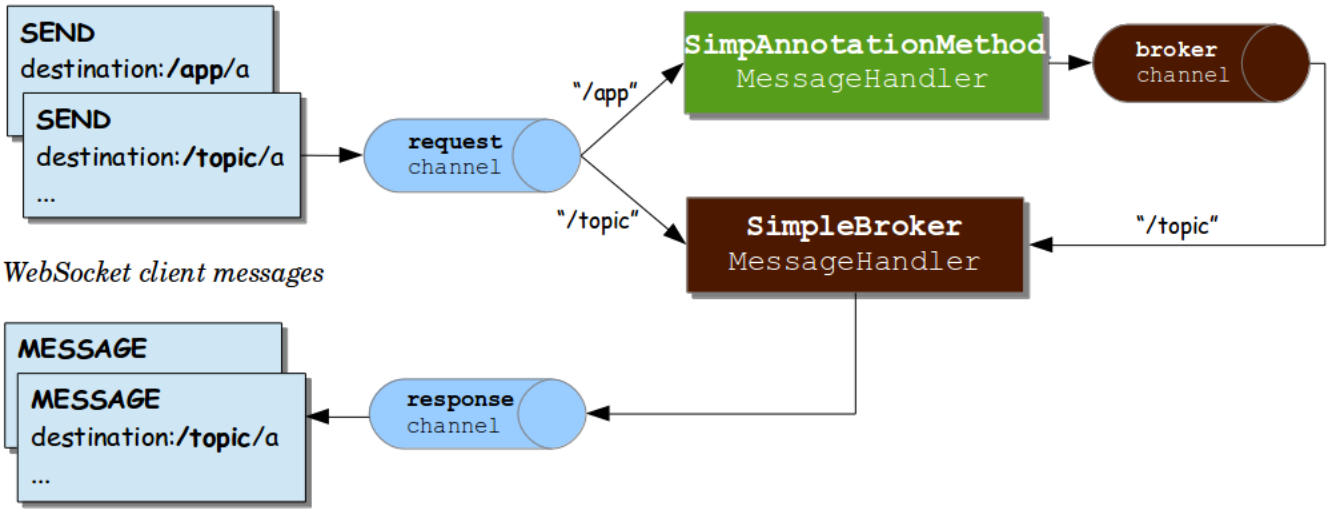
4.4.3. Flow of Messages

When a STOMP endpoint is configured, the Spring application acts as the STOMP broker to connected clients. This section provides a big picture overview of how messages flow within the application.

The `spring-messaging` module provides the foundation for asynchronous message processing. It contains a number of abstractions that originated in the Spring Integration project and are intended for use as building blocks in messaging applications:

- Message — a message with headers and a payload.
- MessageHandler — a contract for handling a message.
- MessageChannel — a contract for sending a message enabling loose coupling between senders and receivers.
- SubscribableChannel — extends `MessageChannel` and sends messages to registered `MessageHandler` subscribers.
- ExecutorSubscribableChannel — a concrete implementation of `SubscribableChannel` that can deliver messages asynchronously via a thread pool.

The `@EnableWebSocketMessageBroker` Java config and the `<websocket:message-broker>` XML config both assemble a concrete message flow. Below is a diagram of the part of the setup when using the simple, in-memory broker:

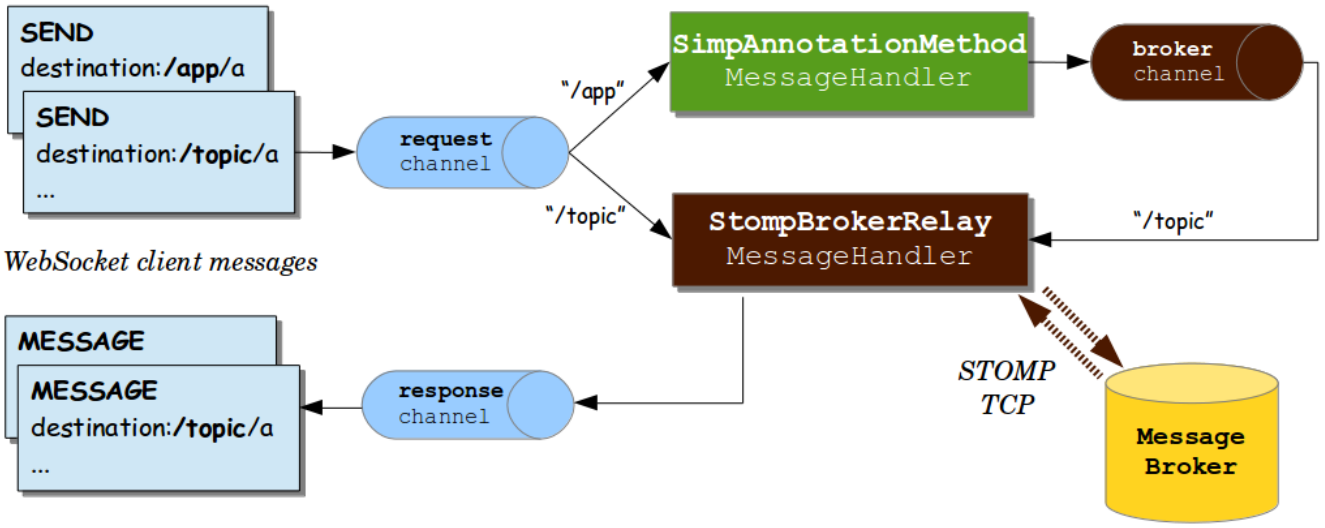


The above setup that includes 3 message channels:

- `"clientInboundChannel"` for messages from WebSocket clients.
- `"clientOutboundChannel"` for messages to WebSocket clients.
- `"brokerChannel"` for messages to the broker from within the application.

The same three channels are also used with a dedicated broker except here a "broker relay" takes

the place of the simple broker:



Messages on the `"clientInboundChannel"` can flow to annotated methods for application handling (e.g. a stock trade execution request) or can be forwarded to the broker (e.g. client subscribing for stock quotes). The STOMP destination is used for simple prefix-based routing. For example the "/app" prefix could route messages to annotated methods while the "/topic" and "/queue" prefixes could route messages to the broker.

When a message-handling annotated method has a return type, its return value is sent as the payload of a Spring `Message` to the `"brokerChannel"`. The broker in turn broadcasts the message to clients. Sending a message to a destination can also be done from anywhere in the application with the help of a messaging template. For example, an HTTP POST handling method can broadcast a message to connected clients, or a service component may periodically broadcast stock quotes.

Below is a simple example to illustrate the flow of messages:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio");
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setApplicationDestinationPrefixes("/app");
        registry.enableSimpleBroker("/topic");
    }

}

@Controller
public class GreetingController {

    @MessageMapping("/greeting") {
    public String handle(String greeting) {
        return "[" + getTimestamp() + ": " + greeting;
    }

}
```

The following explains the message flow for the above example:

- WebSocket clients connect to the WebSocket endpoint at "/portfolio".
- Subscriptions to "/topic/greeting" pass through the "clientInboundChannel" and are forwarded to the broker.
- Greetings sent to "/app/greeting" pass through the "clientInboundChannel" and are forwarded to the `GreetingController`. The controller adds the current time, and the return value is passed through the "brokerChannel" as a message to "/topic/greeting" (destination is selected based on a convention but can be overridden via `@SendTo`).
- The broker in turn broadcasts messages to subscribers, and they pass through the `"clientOutboundChannel"`.

The next section provides more details on annotated methods including the kinds of arguments and return values supported.

4.4.4. Annotation Message Handling

The `@MessageMapping` annotation is supported on methods of `@Controller` classes. It can be used for mapping methods to message destinations and can also be combined with the type-level `@MessageMapping` for expressing shared mappings across all annotated methods within a controller.

By default destination mappings are treated as Ant-style, slash-separated, path patterns, e.g. "/foo*", "/foo/**". etc. They can also contain template variables, e.g. "/foo/{id}" that can then be referenced via `@DestinationVariable`-annotated method arguments.

Applications can also use dot-separated destinations (vs slash). See Using Dot as Separator in @MessageMapping Destinations.

The following method arguments are supported for `@MessageMapping` methods:

- `Message` method argument to get access to the complete message being processed.
- `@Payload`-annotated argument for access to the payload of a message, converted with a `org.springframework.messaging.converter.MessageConverter`. The presence of the annotation is not required since it is assumed by default. Payload method arguments annotated with validation annotations (like `@Validated`) will be subject to JSR-303 validation.
- `@Header`-annotated arguments for access to a specific header value along with type conversion using an `org.springframework.core.convert.converter.Converter` if necessary.
- `@Headers`-annotated method argument that must also be assignable to `java.util.Map` for access to all headers in the message.
- `MessageHeaders` method argument for getting access to a map of all headers.
- `MessageHeaderAccessor`, `SimpMessageHeaderAccessor`, or `StompHeaderAccessor` for access to headers via typed accessor methods.
- `@DestinationVariable`-annotated arguments for access to template variables extracted from the message destination. Values will be converted to the declared method argument type as necessary.
- `java.security.Principal` method arguments reflecting the user logged in at the time of the WebSocket HTTP handshake.

A return value from an `@MessageMapping` method will be converted with a `org.springframework.messaging.converter.MessageConverter` and used as the body of a new message that is then sent, by default, to the `"brokerChannel"` with the same destination as the client message but using the prefix `"/topic"` by default. An `@SendTo` message level annotation can be used to specify any other destination instead. It can also be set a class-level to share a common destination.

A response message may also be provided asynchronously via a `ListenableFuture` or `CompletableFuture`/`CompletionStage` return type signature, analogous to deferred results in an MVC handler method.

A `@SubscribeMapping` annotation can be used to map subscription requests to `@Controller` methods. It is supported on the method level, but can also be combined with a type level `@MessageMapping` annotation that expresses shared mappings across all message handling methods within the same controller.

By default the return value from an `@SubscribeMapping` method is sent as a message directly back to the connected client and does not pass through the broker. This is useful for implementing request-reply message interactions; for example, to fetch application data when the application UI is being initialized. Or alternatively an `@SubscribeMapping` method can be annotated with `@SendTo` in which

case the resulting message is sent to the "brokerChannel" using the specified target destination.

In some cases a controller may need to be decorated with an AOP proxy at runtime. One example is if you choose to have @Transactional annotations directly on the controller. When this is the case, for controllers specifically, we recommend using class-based proxying. This is typically the default choice with controllers. However if a controller must implement an interface that is not a Spring Context callback (e.g. InitializingBean, *Aware, etc), you may need to explicitly configure class-based proxying. For example with <tx:annotation-driven />, change to <tx:annotation-driven proxy-target-class="true" />.

4.4.5. Sending Messages

What if you want to send messages to connected clients from any part of the application? Any application component can send messages to the "brokerChannel". The easiest way to do that is to have a SimpMessagingTemplate injected, and use it to send messages. Typically it should be easy to have it injected by type, for example:

```
@Controller
public class GreetingController {

    private SimpMessagingTemplate template;

    @Autowired
    public GreetingController(SimpMessagingTemplate template) {
        this.template = template;
    }

    @RequestMapping(path="/greetings", method=POST)
    public void greet(String greeting) {
        String text = "[" + getTimestamp() + "]:" + greeting;
        this.template.convertAndSend("/topic/greetings", text);
    }

}
```

But it can also be qualified by its name "brokerMessagingTemplate" if another bean of the same type exists.

4.4.6. Simple Broker

The built-in, simple message broker handles subscription requests from clients, stores them in memory, and broadcasts messages to connected clients with matching destinations. The broker supports path-like destinations, including subscriptions to Ant-style destination patterns.

Applications can also use dot-separated destinations (vs slash). See Using Dot as Separator in @MessageMapping Destinations.

4.4.7. Full-Featured Broker

The simple broker is great for getting started but supports only a subset of STOMP commands (e.g. no acks, receipts, etc.), relies on a simple message sending loop, and is not suitable for clustering. As an alternative, applications can upgrade to using a full-featured message broker.

Check the STOMP documentation for your message broker of choice (e.g. RabbitMQ, ActiveMQ, etc.), install the broker, and run it with STOMP support enabled. Then enable the STOMP broker relay in the Spring configuration instead of the simple broker.

Below is example configuration that enables a full-featured broker:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableStompBrokerRelay("/topic", "/queue");
        registry.setApplicationDestinationPrefixes("/app");
    }

}
```

XML configuration equivalent:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker application-destination-prefix="/app">
        <websocket:stomp-endpoint path="/portfolio" />
            <websocket:sockjs/>
        </websocket:stomp-endpoint>
        <websocket:stomp-broker-relay prefix="/topic,/queue" />
    </websocket:message-broker>

</beans>
```

The "STOMP broker relay" in the above configuration is a Spring MessageHandler that handles messages by forwarding them to an external message broker. To do so it establishes TCP connections to the broker, forwards all messages to it, and then forwards all messages received from the broker to clients through their WebSocket sessions. Essentially it acts as a "relay" that forwards messages in both directions.

Please `org.projectreactor:reactor-net` and `io.netty:netty-all` dependencies to your project for TCP connection management.

Furthermore, application components (e.g. HTTP request handling methods, business services, etc.) can also send messages to the broker relay, as described in Sending Messages, in order to broadcast messages to subscribed WebSocket clients.

In effect, the broker relay enables robust and scalable message broadcasting.

4.4.8. Connections To Full-Featured Broker

A STOMP broker relay maintains a single "system" TCP connection to the broker. This connection is used for messages originating from the server-side application only, not for receiving messages. You can configure the STOMP credentials for this connection, i.e. the STOMP frame `login` and `passcode` headers. This is exposed in both the XML namespace and the Java config as the `systemLogin` /`systemPasscode` properties with default values `guest`/`guest`.

The STOMP broker relay also creates a separate TCP connection for every connected WebSocket client. You can configure the STOMP credentials to use for all TCP connections created on behalf of clients. This is exposed in both the XML namespace and the Java config as the `clientLogin` /`clientPasscode` properties with default values `guest`/`guest`.

The STOMP broker relay always sets the `login` and `passcode` headers on every `CONNECT` frame that it forwards to the broker on behalf of clients. Therefore WebSocket clients need not set those headers; they will be ignored. As the following section explains, instead WebSocket clients should rely on HTTP authentication to protect the WebSocket endpoint and establish the client identity.

The STOMP broker relay also sends and receives heartbeats to and from the message broker over the "system" TCP connection. You can configure the intervals for sending and receiving heartbeats (10 seconds each by default). If connectivity to the broker is lost, the broker relay will continue to try to reconnect, every 5 seconds, until it succeeds.

A Spring bean can implement `ApplicationListener<BrokerAvailabilityEvent>` in order to receive notifications when the "system" connection to the broker is lost and re-established. For example a Stock Quote service broadcasting stock quotes can stop trying to send messages when there is no active "system" connection.

The STOMP broker relay can also be configured with a `virtualHost` property. The value of this property will be set as the `host` header of every `CONNECT` frame and may be useful for example in a cloud environment where the actual host to which the TCP connection is established is different from the host providing the cloud-based STOMP service.

4.4.9. Using Dot as Separator in @MessageMapping Destinations

Although slash-separated path patterns are familiar to web developers, in messaging it is common to use a "." as the separator, for example in the names of topics, queues, exchanges, etc. Applications can also switch to using "." (dot) instead of "/" (slash) as the separator in `@MessageMapping` mappings by configuring a custom `AntPathMatcher`.

In Java config:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

  // ...

  @Override
  public void configureMessageBroker(MessageBrokerRegistry registry) {
    registry.enableStompBrokerRelay("/queue/", "/topic/");
    registry.setApplicationDestinationPrefixes("/app");
    registry.setPathMatcher(new AntPathMatcher("."));
  }

}
```

In XML config:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:websocket="http://www.springframework.org/schema/websocket"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/websocket
    http://www.springframework.org/schema/websocket/spring-websocket.xsd">

  <websocket:message-broker application-destination-prefix="/app" path-matcher=
"pathMatcher">
    <websocket:stomp-endpoint path="/stomp" />
    <websocket:simple-broker prefix="/topic, /queue"/>
  </websocket:message-broker>

  <bean id="pathMatcher" class="org.springframework.util.AntPathMatcher">
    <constructor-arg index="0" value="." />
  </bean>

</beans>
```

And below is a simple example to illustrate a controller with "." separator:

```
@Controller
@MessageMapping("foo")
public class FooController {

  @MessageMapping("bar.{baz}")
  public void handleBaz(@DestinationVariable String baz) {
  }

}
```

If the application prefix is set to "/app" then the foo method is effectively mapped to "/app/foo.bar.{baz}".

4.4.10. Authentication

Every STOMP over WebSocket messaging session begins with an HTTP request — that can be a request to upgrade to WebSockets (i.e. a WebSocket handshake) or in the case of SockJS fallbacks a series of SockJS HTTP transport requests.

Web applications already have authentication and authorization in place to secure HTTP requests. Typically a user is authenticated via Spring Security using some mechanism such as a login page, HTTP basic authentication, or other. The security context for the authenticated user is saved in the HTTP session and is associated with subsequent requests in the same cookie-based session.

Therefore for a WebSocket handshake, or for SockJS HTTP transport requests, typically there will already be an authenticated user accessible via `HttpServletRequest#getUserPrincipal()`. Spring automatically associates that user with a WebSocket or SockJS session created for them and subsequently with all STOMP messages transported over that session through a user header.

In short there is nothing special a typical web application needs to do above and beyond what it already does for security. The user is authenticated at the HTTP request level with a security context maintained through a cookie-based HTTP session which is then associated with WebSocket or SockJS sessions created for that user and results in a user header stamped on every `Message` flowing through the application.

Note that the STOMP protocol does have a "login" and "passcode" headers on the `CONNECT` frame. Those were originally designed for and are still needed for example for STOMP over TCP. However for STOMP over WebSocket by default Spring ignores authorization headers at the STOMP protocol level and assumes the user is already authenticated at the HTTP transport level and expects that the WebSocket or SockJS session contain the authenticated user.

Spring Security provides WebSocket sub-protocol authorization that uses a `ChannelInterceptor` to authorize messages based on the user header in them. Also Spring Session provides a WebSocket integration that ensures the user HTTP session does not expire when the WebSocket session is still active.

4.4.11. Token-based Authentication

Spring Security OAuth provides support for token based security including JSON Web Token (JWT). This can be used as the authentication mechanism in Web applications including STOMP over WebSocket interactions just as described in the previous section, i.e. maintaining identity through a cookie-based session.

At the same time cookie-based sessions are not always the best fit for example in applications that don't wish to maintain a server-side session at all or in mobile applications where it's common to use headers for authentication.

The WebSocket protocol RFC 6455 "doesn't prescribe any particular way that servers can authenticate clients during the WebSocket handshake." In practice however browser clients can only use standard authentication headers (i.e. basic HTTP authentication) or cookies and cannot for example provide custom headers. Likewise the SockJS JavaScript client does not provide a way to send HTTP headers with SockJS transport requests, see sockjs-client issue 196. Instead it does allow sending query parameters that can be used to send a token but that has its own drawbacks, for example as the token may be inadvertently logged with the URL in server logs.

The above limitations are for browser-based clients and do not apply to the Spring Java-based STOMP client which does support sending headers with both WebSocket and SockJS requests.

Therefore applications that wish to avoid the use of cookies may not have any good alternatives for authentication at the HTTP protocol level. Instead of using cookies they may prefer to authenticate with headers at the STOMP messaging protocol level There are 2 simple steps to doing that:

1. Use the STOMP client to pass authentication header(s) at connect time.
2. Process the authentication header(s) with a `ChannelInterceptor`.

Below is the example server-side configuration to register a custom authentication interceptor. Note that an interceptor only needs to authenticate and set the user header on the CONNECT `Message`. Spring will note and save the authenticated user and associate it with subsequent STOMP messages on the same session:

```
@Configuration
@EnableWebSocketMessageBroker
public class MyConfig extends AbstractWebSocketMessageBrokerConfigurer {

  @Override
  public void configureClientInboundChannel(ChannelRegistration registration) {
    registration.setInterceptors(new ChannelInterceptorAdapter() {

        @Override
        public Message<?> preSend(Message<?> message, MessageChannel channel) {

            StompHeaderAccessor accessor =
                MessageHeaderAccessor.getAccessor(message, StompHeaderAccessor.class);

            if (StompCommand.CONNECT.equals(accessor.getCommand())) {
                Authentication user = ... ; // access authentication header(s)
                accessor.setUser(user);
            }

            return message;
        }
    });
  }
}
```

Also note that when using Spring Security's authorization for messages, at present you will need to ensure that the authentication `ChannelInterceptor` config is ordered ahead of Spring Security's. This is best done by declaring the custom interceptor in its own sub-class of `AbstractWebSocketMessageBrokerConfigurer` marked with `@Order(Ordered.HIGHEST_PRECEDENCE + 99)`.

4.4.12. User Destinations

An application can send messages targeting a specific user, and Spring's STOMP support recognizes destinations prefixed with `"/user/"` for this purpose. For example, a client might subscribe to the destination `"/user/queue/position-updates"`. This destination will be handled by the `UserDestinationMessageHandler` and transformed into a destination unique to the user session, e.g. `"/queue/position-updates-user123"`. This provides the convenience of subscribing to a generically named destination while at the same time ensuring no collisions with other users subscribing to the same destination so that each user can receive unique stock position updates.

On the sending side messages can be sent to a destination such as `"/user/{username}/queue/position-updates"`, which in turn will be translated by the `UserDestinationMessageHandler` into one or more destinations, one for each session associated with the user. This allows any component within the application to send messages targeting a specific user without necessarily knowing anything more than their name and the generic destination. This is also supported through an annotation as well as a messaging template.

For example, a message-handling method can send messages to the user associated with the message being handled through the `@SendToUser` annotation (also supported on the class-level to

share a common destination):

```
@Controller
public class PortfolioController {

    @MessageMapping("/trade")
    @SendToUser("/queue/position-updates")
    public TradeResult executeTrade(Trade trade, Principal principal) {
        // ...
        return tradeResult;
    }
}
```

If the user has more than one session, by default all of the sessions subscribed to the given destination are targeted. However sometimes, it may be necessary to target only the session that sent the message being handled. This can be done by setting the `broadcast` attribute to false, for example:

```
@Controller
public class MyController {

    @MessageMapping("/action")
    public void handleAction() throws Exception{
        // raise MyBusinessException here
    }

    @MessageExceptionHandler
    @SendToUser(destinations="/queue/errors", broadcast=false)
    public ApplicationError handleException(MyBusinessException exception) {
        // ...
        return appError;
    }
}
```

While user destinations generally imply an authenticated user, it isn't required strictly. A WebSocket session that is not associated with an authenticated user can subscribe to a user destination. In such cases the `@SendToUser` annotation will behave exactly the same as with `broadcast=false`, i.e. targeting only the session that sent the message being handled.

It is also possible to send a message to user destinations from any application component by injecting the `SimpMessagingTemplate` created by the Java config or XML namespace, for example (the bean name is `"brokerMessagingTemplate"` if required for qualification with `@Qualifier`):

```
@Service
public class TradeServiceImpl implements TradeService {

    private final SimpMessagingTemplate messagingTemplate;

    @Autowired
    public TradeServiceImpl(SimpMessagingTemplate messagingTemplate) {
        this.messagingTemplate = messagingTemplate;
    }

    // ...

    public void afterTradeExecuted(Trade trade) {
        this.messagingTemplate.convertAndSendToUser(
                trade.getUserName(), "/queue/position-updates", trade.getResult());
    }
}
```

When using user destinations with an external message broker, check the broker documentation on how to manage inactive queues, so that when the user session is over, all unique user queues are removed. For example, RabbitMQ creates auto-delete queues when destinations like `/exchange/amq.direct/position-updates` are used. So in that case the client could subscribe to `/user/exchange/amq.direct/position-updates`. Similarly, ActiveMQ has configuration options for purging inactive destinations.

In a multi-application server scenario a user destination may remain unresolved because the user is connected to a different server. In such cases you can configure a destination to broadcast unresolved messages to so that other servers have a chance to try. This can be done through the `userDestinationBroadcast` property of the `MessageBrokerRegistry` in Java config and the `user-destination-broadcast` attribute of the `message-broker` element in XML.

4.4.13. Listening To ApplicationContext Events and Intercepting Messages

Several `ApplicationContext` events (listed below) are published and can be received by implementing Spring's `ApplicationListener` interface.

- `BrokerAvailabilityEvent` — indicates when the broker becomes available/unavailable. While the "simple" broker becomes available immediately on startup and remains so while the application is running, the STOMP "broker relay" may lose its connection to the full featured broker, for example if the broker is restarted. The broker relay has reconnect logic and will re-establish the "system" connection to the broker when it comes back, hence this event is published whenever the state changes from connected to disconnected and vice versa. Components using the `SimpMessagingTemplate` should subscribe to this event and avoid sending messages at times when the broker is not available. In any case they should be prepared to handle `MessageDeliveryException` when sending a message.
- `SessionConnectEvent` — published when a new STOMP CONNECT is received indicating the start

of a new client session. The event contains the message representing the connect including the session id, user information (if any), and any custom headers the client may have sent. This is useful for tracking client sessions. Components subscribed to this event can wrap the contained message using `SimpMessageHeaderAccessor` or `StompMessageHeaderAccessor`.

- `SessionConnectedEvent` — published shortly after a `SessionConnectEvent` when the broker has sent a STOMP CONNECTED frame in response to the CONNECT. At this point the STOMP session can be considered fully established.
- `SessionSubscribeEvent` — published when a new STOMP SUBSCRIBE is received.
- `SessionUnsubscribeEvent` — published when a new STOMP UNSUBSCRIBE is received.
- `SessionDisconnectEvent` — published when a STOMP session ends. The DISCONNECT may have been sent from the client, or it may also be automatically generated when the WebSocket session is closed. In some cases this event may be published more than once per session. Components should be idempotent with regard to multiple disconnect events.

When using a full-featured broker, the STOMP "broker relay" automatically reconnects the "system" connection in case the broker becomes temporarily unavailable. Client connections however are not automatically reconnected. Assuming heartbeats are enabled, the client will typically notice the broker is not responding within 10 seconds. Clients need to implement their own reconnect logic.

Furthermore, an application can directly intercept every incoming and outgoing message by registering a `ChannelInterceptor` on the respective message channel. For example to intercept inbound messages:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

  @Override
  public void configureClientInboundChannel(ChannelRegistration registration) {
    registration.setInterceptors(new MyChannelInterceptor());
  }
}
```

A custom `ChannelInterceptor` can extend the empty method base class `ChannelInterceptorAdapter` and use `StompHeaderAccessor` or `SimpMessageHeaderAccessor` to access information about the message.

```
public class MyChannelInterceptor extends ChannelInterceptorAdapter {

  @Override
  public Message<?> preSend(Message<?> message, MessageChannel channel) {
    StompHeaderAccessor accessor = StompHeaderAccessor.wrap(message);
    StompCommand command = accessor.getStompCommand();
    // ...
    return message;
  }
}
```

4.4.14. STOMP Client

Spring provides a STOMP over WebSocket client and a STOMP over TCP client.

To begin create and configure `WebSocketStompClient`:

```
WebSocketClient webSocketClient = new StandardWebSocketClient();
WebSocketStompClient stompClient = new WebSocketStompClient(webSocketClient);
stompClient.setMessageConverter(new StringMessageConverter());
stompClient.setTaskScheduler(taskScheduler); // for heartbeats
```

In the above example `StandardWebSocketClient` could be replaced with `SockJsClient` since that is also an implementation of `WebSocketClient`. The `SockJsClient` can use WebSocket or HTTP-based transport as a fallback. For more details see SockJS Client.

Next establish a connection and provide a handler for the STOMP session:

```
String url = "ws://127.0.0.1:8080/endpoint";
StompSessionHandler sessionHandler = new MyStompSessionHandler();
stompClient.connect(url, sessionHandler);
```

When the session is ready for use the handler is notified:

```
public class MyStompSessionHandler extends StompSessionHandlerAdapter {

    @Override
    public void afterConnected(StompSession session, StompHeaders connectedHeaders) {
        // ...
    }
}
```

Once the session is established any payload can be sent and that will be serialized with the configured `MessageConverter`:

```
session.send("/topic/foo", "payload");
```

You can also subscribe to destinations. The `subscribe` methods require a handler for messages on the subscription and return a `Subscription` handle that can be used to unsubscribe. For each received message the handler can specify the target Object type the payload should be deserialized to:

```
session.subscribe("/topic/foo", new StompFrameHandler() {

    @Override
    public Type getPayloadType(StompHeaders headers) {
        return String.class;
    }

    @Override
    public void handleFrame(StompHeaders headers, Object payload) {
        // ...
    }

});
```

To enable STOMP heartbeat configure `WebSocketStompClient` with a `TaskScheduler` and optionally customize the heartbeat intervals, 10 seconds for write inactivity which causes a heartbeat to be sent and 10 seconds for read inactivity which closes the connection.

When using `WebSocketStompClient` for performance tests to simulate thousands of clients from the same machine consider turning off heartbeats since each connection schedules its own heartbeat tasks and that's not optimized for a a large number of clients running on the same machine.

The STOMP protocol also supports receipts where the client must add a "receipt" header to which the server responds with a RECEIPT frame after the send or subscribe are processed. To support this the `StompSession` offers `setAutoReceipt(boolean)` that causes a "receipt" header to be added on every subsequent send or subscribe. Alternatively you can also manually add a "receipt" header to the `StompHeaders`. Both send and subscribe return an instance of `Receiptable` that can be used to register for receipt success and failure callbacks. For this feature the client must be configured with a `TaskScheduler` and the amount of time before a receipt expires (15 seconds by default).

Note that `StompSessionHandler` itself is a `StompFrameHandler` which allows it to handle ERROR frames in addition to the `handleException` callback for exceptions from the handling of messages, and `handleTransportError` for transport-level errors including `ConnectionLostException`.

4.4.15. WebSocket Scope

Each WebSocket session has a map of attributes. The map is attached as a header to inbound client messages and may be accessed from a controller method, for example:

```
@Controller
public class MyController {

    @MessageMapping("/action")
    public void handle(SimpMessageHeaderAccessor headerAccessor) {
        Map<String, Object> attrs = headerAccessor.getSessionAttributes();
        // ...
    }
}
```

It is also possible to declare a Spring-managed bean in the `websocket` scope. WebSocket-scoped beans can be injected into controllers and any channel interceptors registered on the "clientInboundChannel". Those are typically singletons and live longer than any individual WebSocket session. Therefore you will need to use a scope proxy mode for WebSocket-scoped beans:

```
@Component
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class MyBean {

    @PostConstruct
    public void init() {
        // Invoked after dependencies injected
    }

    // ...

    @PreDestroy
    public void destroy() {
        // Invoked when the WebSocket session ends
    }
}

@Controller
public class MyController {

    private final MyBean myBean;

    @Autowired
    public MyController(MyBean myBean) {
        this.myBean = myBean;
    }

    @MessageMapping("/action")
    public void handle() {
        // this.myBean from the current WebSocket session
    }
}
```

As with any custom scope, Spring initializes a new `MyBean` instance the first time it is accessed from the controller and stores the instance in the WebSocket session attributes. The same instance is returned subsequently until the session ends. WebSocket-scoped beans will have all Spring lifecycle methods invoked as shown in the examples above.

4.4.16. Configuration and Performance

There is no silver bullet when it comes to performance. Many factors may affect it including the size of messages, the volume, whether application methods perform work that requires blocking, as well as external factors such as network speed and others. The goal of this section is to provide an overview of the available configuration options along with some thoughts on how to reason about scaling.

In a messaging application messages are passed through channels for asynchronous executions backed by thread pools. Configuring such an application requires good knowledge of the channels and the flow of messages. Therefore it is recommended to review Flow of Messages.

The obvious place to start is to configure the thread pools backing the `"clientInboundChannel"` and the `"clientOutboundChannel"`. By default both are configured at twice the number of available processors.

If the handling of messages in annotated methods is mainly CPU bound then the number of threads for the `"clientInboundChannel"` should remain close to the number of processors. If the work they do is more IO bound and requires blocking or waiting on a database or other external system then the thread pool size will need to be increased.

`ThreadPoolExecutor` has 3 important properties. Those are the core and the max thread pool size as well as the capacity for the queue to store tasks for which there are no available threads.

A common point of confusion is that configuring the core pool size (e.g. 10) and max pool size (e.g. 20) results in a thread pool with 10 to 20 threads. In fact if the capacity is left at its default value of Integer.MAX_VALUE then the thread pool will never increase beyond the core pool size since all additional tasks will be queued.

Please review the Javadoc of `ThreadPoolExecutor` to learn how these properties work and understand the various queuing strategies.

On the `"clientOutboundChannel"` side it is all about sending messages to WebSocket clients. If clients are on a fast network then the number of threads should remain close to the number of available processors. If they are slow or on low bandwidth they will take longer to consume messages and put a burden on the thread pool. Therefore increasing the thread pool size will be necessary.

While the workload for the "clientInboundChannel" is possible to predict — after all it is based on what the application does — how to configure the "clientOutboundChannel" is harder as it is based on factors beyond the control of the application. For this reason there are two additional properties related to the sending of messages. Those are the `"sendTimeLimit"` and the `"sendBufferSizeLimit"`. Those are used to configure how long a send is allowed to take and how much data can be buffered when sending messages to a client.

The general idea is that at any given time only a single thread may be used to send to a client. All additional messages meanwhile get buffered and you can use these properties to decide how long sending a message is allowed to take and how much data can be buffered in the mean time. Please review the Javadoc and documentation of the XML schema for this configuration for important additional details.

Here is example configuration:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureWebSocketTransport(WebSocketTransportRegistration
registration) {
        registration.setSendTimeLimit(15 * 1000).setSendBufferSizeLimit(512 * 1024);
    }

    // ...

}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker>
        <websocket:transport send-timeout="15000" send-buffer-size="524288" />
        <!-- ... -->
    </websocket:message-broker>

</beans>
```

The WebSocket transport configuration shown above can also be used to configure the maximum allowed size for incoming STOMP messages. Although in theory a WebSocket message can be almost unlimited in size, in practice WebSocket servers impose limits — for example, 8K on Tomcat and 64K on Jetty. For this reason STOMP clients such as stomp.js split larger STOMP messages at 16K boundaries and send them as multiple WebSocket messages thus requiring the server to buffer and re-assemble.

Spring's STOMP over WebSocket support does this so applications can configure the maximum size for STOMP messages irrespective of WebSocket server specific message sizes. Do keep in mind that the WebSocket message size will be automatically adjusted if necessary to ensure they can carry

16K WebSocket messages at a minimum.

Here is example configuration:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureWebSocketTransport(WebSocketTransportRegistration
registration) {
        registration.setMessageSizeLimit(128 * 1024);
    }

    // ...

}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker>
        <websocket:transport message-size="131072" />
        <!-- ... -->
    </websocket:message-broker>

</beans>
```

An important point about scaling is using multiple application instances. Currently it is not possible to do that with the simple broker. However when using a full-featured broker such as RabbitMQ, each application instance connects to the broker and messages broadcast from one application instance can be broadcast through the broker to WebSocket clients connected through any other application instances.

4.4.17. Runtime Monitoring

When using `@EnableWebSocketMessageBroker` or `<websocket:message-broker>` key infrastructure components automatically gather stats and counters that provide important insight into the internal state of the application. The configuration also declares a bean of type `WebSocketMessageBrokerStats` that gathers all available information in one place and by default logs it at `INFO` level once every 30 minutes. This bean can be exported to JMX through Spring's `MBeanExporter` for viewing at runtime, for example through JDK's `jconsole`. Below is a summary of

the available information.

Client WebSocket Sessions

Current

indicates how many client sessions there are currently with the count further broken down by WebSocket vs HTTP streaming and polling SockJS sessions.

Total

indicates how many total sessions have been established.

Abnormally Closed

Connect Failures

these are sessions that got established but were closed after not having received any messages within 60 seconds. This is usually an indication of proxy or network issues.

Send Limit Exceeded

sessions closed after exceeding the configured send timeout or the send buffer limits which can occur with slow clients (see previous section).

Transport Errors

sessions closed after a transport error such as failure to read or write to a WebSocket connection or HTTP request/response.

STOMP Frames

the total number of CONNECT, CONNECTED, and DISCONNECT frames processed indicating how many clients connected on the STOMP level. Note that the DISCONNECT count may be lower when sessions get closed abnormally or when clients close without sending a DISCONNECT frame.

STOMP Broker Relay

TCP Connections

indicates how many TCP connections on behalf of client WebSocket sessions are established to the broker. This should be equal to the number of client WebSocket sessions + 1 additional shared "system" connection for sending messages from within the application.

STOMP Frames

the total number of CONNECT, CONNECTED, and DISCONNECT frames forwarded to or received from the broker on behalf of clients. Note that a DISCONNECT frame is sent to the broker regardless of how the client WebSocket session was closed. Therefore a lower DISCONNECT frame count is an indication that the broker is pro-actively closing connections, may be because of a heartbeat that didn’t arrive in time, an invalid input frame, or other.

Client Inbound Channel

stats from thread pool backing the "clientInboundChannel" providing insight into the health of incoming message processing. Tasks queueing up here is an indication the application may be too slow to handle messages. If there I/O bound tasks (e.g. slow database query, HTTP request to 3rd party REST API, etc) consider increasing the thread pool size.

Client Outbound Channel

stats from the thread pool backing the "clientOutboundChannel" providing insight into the health of broadcasting messages to clients. Tasks queueing up here is an indication clients are too slow to consume messages. One way to address this is to increase the thread pool size to accommodate the number of concurrent slow clients expected. Another option is to reduce the send timeout and send buffer size limits (see the previous section).

SockJS Task Scheduler

stats from thread pool of the SockJS task scheduler which is used to send heartbeats. Note that when heartbeats are negotiated on the STOMP level the SockJS heartbeats are disabled.

4.4.18. Testing Annotated Controller Methods

There are two main approaches to testing applications using Spring's STOMP over WebSocket support. The first is to write server-side tests verifying the functionality of controllers and their annotated message handling methods. The second is to write full end-to-end tests that involve running a client and a server.

The two approaches are not mutually exclusive. On the contrary each has a place in an overall test strategy. Server-side tests are more focused and easier to write and maintain. End-to-end integration tests on the other hand are more complete and test much more, but they're also more involved to write and maintain.

The simplest form of server-side tests is to write controller unit tests. However this is not useful enough since much of what a controller does depends on its annotations. Pure unit tests simply can't test that.

Ideally controllers under test should be invoked as they are at runtime, much like the approach to testing controllers handling HTTP requests using the Spring MVC Test framework. i.e. without running a Servlet container but relying on the Spring Framework to invoke the annotated controllers. Just like with Spring MVC Test here there are two two possible alternatives, either using a "context-based" or "standalone" setup:

1. Load the actual Spring configuration with the help of the Spring TestContext framework, inject "clientInboundChannel" as a test field, and use it to send messages to be handled by controller methods.
2. Manually set up the minimum Spring framework infrastructure required to invoke controllers (namely the `SimpAnnotationMethodMessageHandler`) and pass messages for controllers directly to it.

Both of these setup scenarios are demonstrated in the tests for the stock portfolio sample application.

The second approach is to create end-to-end integration tests. For that you will need to run a WebSocket server in embedded mode and connect to it as a WebSocket client sending WebSocket messages containing STOMP frames. The tests for the stock portfolio sample application also demonstrates this approach using Tomcat as the embedded WebSocket server and a simple STOMP client for test purposes.

Chapter 5. Spring WebFlux framework

This section provides basic information on the reactive programming support for Web applications in Spring Framework 5.

5.1. Introduction

5.1.1. What is Reactive Programming?

In plain terms reactive programming is about non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale vertically (i.e. within the JVM) rather than horizontally (i.e. through clustering).

A key aspect of reactive applications is the concept of backpressure which is a mechanism to ensure producers don't overwhelm consumers. For example in a pipeline of reactive components extending from the database to the HTTP response when the HTTP connection is too slow the data repository can also slow down or stop completely until network capacity frees up.

Reactive programming also leads to a major shift from imperative to declarative async composition of logic. It is comparable to writing blocking code vs using the `CompletableFuture` from Java 8 to compose follow-up actions via lambda expressions.

For a longer introduction check the blog series "Notes on Reactive Programming" by Dave Syer.

5.1.2. Reactive API and Building Blocks

Spring Framework 5 embraces Reactive Streams as the contract for communicating backpressure across async components and libraries. Reactive Streams is a specification created through industry collaboration that has also been adopted in Java 9 as `java.util.concurrent.Flow`.

The Spring Framework uses Reactor internally for its own reactive support. Reactor is a Reactive Streams implementation that further extends the basic Reactive Streams `Publisher` contract with the `Flux` and `Mono` composable API types to provide declarative operations on data sequences of `0..N` and `0..1`.

The Spring Framework exposes `Flux` and `Mono` in many of its own reactive APIs. At the application level however, as always, Spring provides choice and fully supports the use of RxJava. For more on reactive types check the post "Understanding Reactive Types" by Sebastien Deleuze.

5.2. Spring WebFlux Module

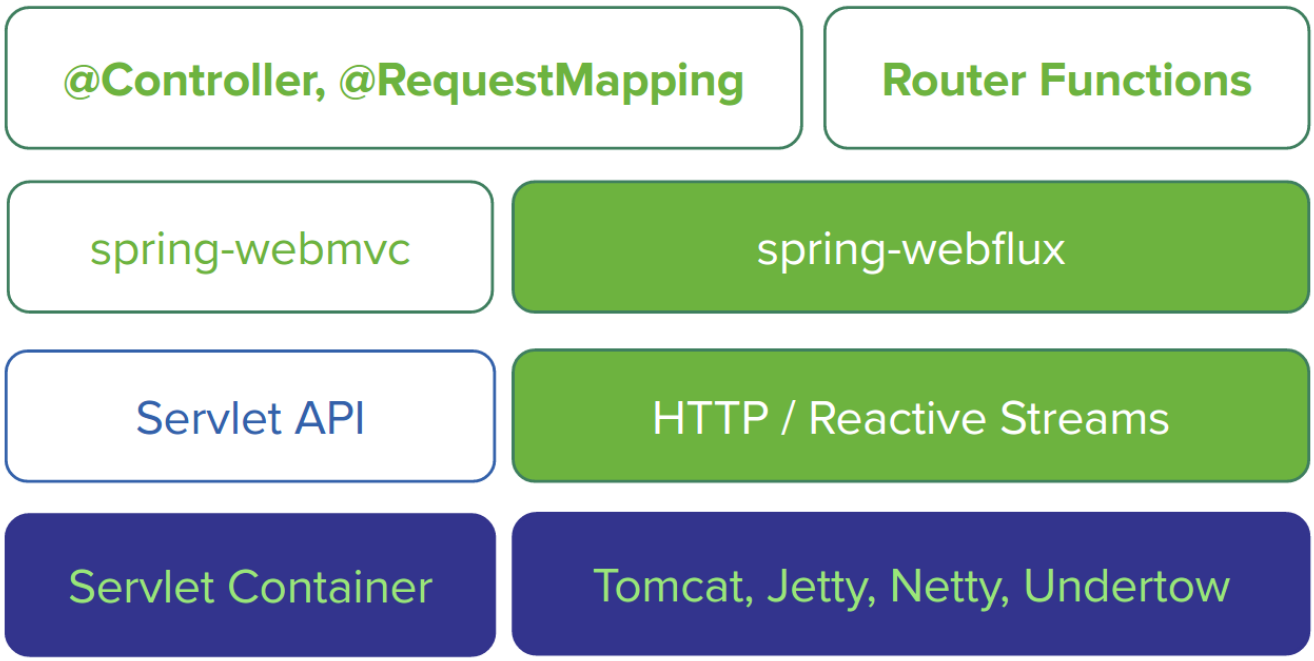
Spring Framework 5 includes a new `spring-webflux` module. The module contains support for reactive HTTP and WebSocket clients as well as for reactive server web applications including REST, HTML browser, and WebSocket style interactions.

5.2.1. Server Side

On the server-side WebFlux supports 2 distinct programming models:

- Annotation-based with `@Controller` and the other annotations supported also with Spring MVC
- Functional, Java 8 lambda style routing and handling

Both programming models are executed on the same reactive foundation that adapts non-blocking HTTP runtimes to the Reactive Streams API. The diagram below shows the server-side stack including traditional, Servlet-based Spring MVC on the left from the `spring-webmvc` module and also the reactive stack on the right from the `spring-webflux` module.



WebFlux can run on Servlet containers with support for the Servlet 3.1 Non-Blocking IO API as well as on other async runtimes such as Netty and Undertow. Each runtime is adapted to a reactive `ServerHttpRequest` and `ServerHttpResponse` exposing the body of the request and response as `Flux<DataBuffer>`, rather than `InputStream` and `OutputStream`, with reactive backpressure. REST-style JSON and XML serialization and deserialization is supported on top as a `Flux<Object>`, and so is HTML view rendering and Server-Sent Events.

Annotation-based Programming Model

The same `@Controller` programming model and the same annotations used in Spring MVC are also supported in WebFlux. The main difference is that the underlying core, framework contracts — i.e. `HandlerMapping`, `HandlerAdapter`, are non-blocking and operate on the reactive `ServerHttpRequest` and `ServerHttpResponse` rather than on the `HttpServletRequest` and `HttpServletResponse`. Below is an example with a reactive controller:

```
@RestController
public class PersonController {

    private final PersonRepository repository;

    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }

    @PostMapping("/person")
    Mono<Void> create(@RequestBody Publisher<Person> personStream) {
        return this.repository.save(personStream).then();
    }

    @GetMapping("/person")
    Flux<Person> list() {
        return this.repository.findAll();
    }

    @GetMapping("/person/{id}")
    Mono<Person> findById(@PathVariable String id) {
        return this.repository.findOne(id);
    }
}
```

Functional Programming Model

5.3. Spring WebFlux Functional Programming Model

5.3.1. HandlerFunctions

Incoming HTTP requests are handled by a `HandlerFunction`, which is essentially a function that takes a `ServerRequest` and returns a `Mono<ServerResponse>`. The annotation counterpart to a handler function would be a method with `@RequestMapping`.

`ServerRequest` and `ServerResponse` are immutable interfaces that offer JDK-8 friendly access to the underlying HTTP messages. Both are fully reactive by building on top of Reactor: the request expose the body as `Flux` or `Mono`; the response accepts any Reactive Streams `Publisher` as body.

`ServerRequest` gives access to various HTTP request elements: the method, URI, query parameters, and — through the separate `ServerRequest.Headers` interface — the headers. Access to the body is provided through the `body` methods. For instance, this is how to extract the request body into a `Mono<String>`:

```
Mono<String> string = request.bodyToMono(String.class);
```

And here is how to extract the body into a `Flux`, where `Person` is a class that can be deserialised from

the contents of the body (i.e. `Person` is supported by Jackson if the body contains JSON, or JAXB if XML).

```
Flux<Person> people = request.bodyToFlux(Person.class);
```

The two methods above (`bodyToMono` and `bodyToFlux`) are, in fact, convenience methods that use the generic `ServerRequest.body(BodyExtractor)` method. `BodyExtractor` is a functional strategy interface that allows you to write your own extraction logic, but common `BodyExtractor` instances can be found in the `BodyExtractors` utility class. So, the above examples can be replaced with:

```
Mono<String> string = request.body(BodyExtractors.toMono(String.class);
Flux<Person> people = request.body(BodyExtractors.toFlux(Person.class);
```

Similarly, `ServerResponse` provides access to the HTTP response. Since it is immutable, you create a `ServerResponse` with a builder. The builder allows you to set the response status, add response headers, and provide a body. For instance, this is how to create a response with a 200 OK status, a JSON content-type, and a body:

```
Mono<Person> person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person);
```

And here is how to build a response with a 201 Created status, Location header, and empty body:

```
URI location = ...
ServerResponse.created(location).build();
```

Putting these together allows us to create a `HandlerFunction`. For instance, here is an example of a simple "Hello World" handler lambda, that returns a response with a 200 status and a body based on a String:

```
HandlerFunction<ServerResponse> helloWorld =
  request -> ServerResponse.ok().body(fromObject("Hello World"));
```

Writing handler functions as lambda's, as we do above, is convenient, but perhaps lacks in readability and becomes less maintainable when dealing with multiple functions. Therefore, it is recommended to group related handler functions into a handler or controller class. For example, here is a class that exposes a reactive `Person` repository:

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.fromObject;

public class PersonHandler {

    private final PersonRepository repository;

    public PersonHandler(PersonRepository repository) {
        this.repository = repository;
    }

    public Mono<ServerResponse> listPeople(ServerRequest request) { ①
        Flux<Person> people = repository.allPeople();
        return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person
.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) { ②
        Mono<Person> person = request.bodyToMono(Person.class);
        return ServerResponse.ok().build(repository.savePerson(person));
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) { ③
        int personId = Integer.valueOf(request.pathVariable("id"));
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();
        Mono<Person> personMono = this.repository.getPerson(personId);
        return personMono
                .flatMap(person -> ServerResponse.ok().contentType(APPLICATION_JSON)
.body(fromObject(person)))
                .switchIfEmpty(notFound);
    }
}
```

① `listPeople` is a handler function that returns all `Person` objects found in the repository as JSON.

② `createPerson` is a handler function that stores a new `Person` contained in the request body. Note that `PersonRepository.savePerson(Person)` returns `Mono<Void>`: an empty Mono that emits a completion signal when the person has been read from the request and stored. So we use the `build(Publisher<Void>)` method to send a response when that completion signal is received, i.e. when the `Person` has been saved.

③ `getPerson` is a handler function that returns a single person, identified via the path variable `id`. We retrieve that `Person` via the repository, and create a JSON response if it is found. If it is not found, we use `switchIfEmpty(Mono<T>)` to return a 404 Not Found response.

5.3.2. RouterFunctions

Incoming requests are routed to handler functions with a **`RouterFunction`**, which is a function that takes a `ServerRequest`, and returns a `Mono<HandlerFunction>`. If a request matches a particular route, a handler function is returned; otherwise it returns an empty `Mono`. The `RouterFunction` has a similar

purpose as the `@RequestMapping` annotation in `@Controller` classes.

Typically, you do not write router functions yourself, but rather use `RouterFunctions.route(RequestPredicate, HandlerFunction)` to create one using a request predicate and handler function. If the predicate applies, the request is routed to the given handler function; otherwise no routing is performed, resulting in a 404 Not Found response. Though you can write your own `RequestPredicate`, you do not have to: the `RequestPredicates` utility class offers commonly used predicates, such matching based on path, HTTP method, content-type, etc. Using `route`, we can route to our "Hello World" handler function:

```
RouterFunction<ServerResponse> helloWorldRoute =
    RouterFunctions.route(RequestPredicates.path("/hello-world"),
    request -> Response.ok().body(fromObject("Hello World")));
```

Two router functions can be composed into a new router function that routes to either handler function: if the predicate of the first route does not match, the second is evaluated. Composed router functions are evaluated in order, so it makes sense to put specific functions before generic ones. You can compose two router functions by calling `RouterFunction.and(RouterFunction)`, or by calling `RouterFunction.andRoute(RequestPredicate, HandlerFunction)`, which is a convenient combination of `RouterFunction.and()` with `RouterFunctions.route()`.

Given the `PersonHandler` we showed above, we can now define a router function that routes to the respective handler functions. We use method-references to refer to the handler functions:

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> personRoute =
    route(GET("/person/{id}").and(accept(APPLICATION_JSON)), handler::getPerson)
        .andRoute(GET("/person").and(accept(APPLICATION_JSON)), handler::listPeople)
        .andRoute(POST("/person").and(contentType(APPLICATION_JSON)), handler:
:createPerson);
```

Besides router functions, you can also compose request predicates, by calling `RequestPredicate.and(RequestPredicate)` or `RequestPredicate.or(RequestPredicate)`. These work as expected: for `and` the resulting predicate matches if **both** given predicates match; `or` matches if **either** predicate does. Most of the predicates found in `RequestPredicates` are compositions. For instance, `RequestPredicates.GET(String)` is a composition of `RequestPredicates.method(HttpMethod)` and `RequestPredicates.path(String)`.

Running a Server

Now there is just one piece of the puzzle missing: running a router function in an HTTP server. You can convert a router function into a `HttpHandler` by using `RouterFunctions.toHttpHandler(RouterFunction)`. The `HttpHandler` allows you to run on a wide

variety of reactive runtimes: Reactor Netty, Servlet 3.1+, and Undertow. Here is how we run a router function in Reactor Netty, for instance:

```
RouterFunction<ServerResponse> route = ...
HttpHandler httpHandler = RouterFunctions.toHttpHandler(route);
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(httpHandler);
HttpServer server = HttpServer.create(HOST, PORT);
server.newHandler(adapter).block();
```

For Tomcat it looks like this:

```
RouterFunction<ServerResponse> route = ...
HttpHandler httpHandler = RouterFunctions.toHttpHandler(route);
HttpServlet servlet = new ServletHttpHandlerAdapter(httpHandler);
Tomcat server = new Tomcat();
Context rootContext = server.addContext("", System.getProperty("java.io.tmpdir"));
Tomcat.addServlet(rootContext, "servlet", servlet);
rootContext.addServletMapping("/", "servlet");
tomcatServer.start();
```

5.3.3. HandlerFilterFunction

Routes mapped by a router function can be filtered by calling `RouterFunction.filter(HandlerFilterFunction)`, where `HandlerFilterFunction` is essentially a function that takes a `ServerRequest` and `HandlerFunction`, and returns a `ServerResponse`. The handler function parameter represents the next element in the chain: this is typically the `HandlerFunction` that is routed to, but can also be another `FilterFunction` if multiple filters are applied. With annotations, similar functionality can be achieved using `@ControllerAdvice` and/or a `ServletFilter`. Let's add a simple security filter to our route, assuming that we have a `SecurityManager` that can determine whether a particular path is allowed:

```
import static org.springframework.http.HttpStatus.UNAUTHORIZED;

SecurityManager securityManager = ...
RouterFunction<ServerResponse> route = ...

RouterFunction<ServerResponse> filteredRoute =
    route.filter(request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
  });
```

You can see in this example that invoking the `next.handle(ServerRequest)` is optional: we only allow

the handler function to be executed when access is allowed.

5.3.4. Client Side

WebFlux includes a functional, reactive `WebClient` that offers a fully non-blocking and reactive alternative to the `RestTemplate`. It exposes network input and output as a reactive `ClientHttpRequest` and `ClientHttpResponse` where the body of the request and response is a `Flux<DataBuffer>` rather than an `InputStream` and `OutputStream`. In addition it supports the same reactive JSON, XML, and SSE serialization mechanism as on the server side so you can work with typed objects. Below is an example of using the `WebClient` which requires a `ClientHttpConnector` implementation to plug in a specific HTTP client such as Reactor Netty:

```
WebClient client = WebClient.create("http://example.com");

Mono<Account> account = client.get()
        .url("/accounts/{id}", 1L)
        .accept(APPLICATION_JSON)
        .exchange(request)
        .flatMap(response -> response.bodyToMono(Account.class));
```

The `AsyncRestTemplate` also supports non-blocking interactions. The main difference is it can't support non-blocking streaming, like for example Twitter one, because fundamentally it's still based and relies on `InputStream` and `OutputStream`.

5.3.5. Request and Response Body Conversion

The `spring-core` module provides reactive `Encoder` and `Decoder` contracts that enable the serialization of a `Flux` of bytes to and from typed objects. The `spring-web` module adds JSON (Jackson) and XML (JAXB) implementations for use in web applications as well as others for SSE streaming and zero-copy file transfer.

The following Reactive APIs are supported:

- Reactor 3.x is supported out of the box
- RxJava 2.x is supported when `io.reactivex.rxjava2:rxjava` dependency is on the classpath
- RxJava 1.x is supported when both `io.reactivex:rxjava` and `io.reactivex:rxjava-reactive-streams` (adapter between RxJava and Reactive Streams) dependencies are on the classpath

For example the request body can be one of the following way and it will be decoded automatically in both the annotation and the functional programming models:

- `Account account` — the account is deserialized without blocking before the controller is invoked.
- `Mono<Account> account` — the controller can use the `Mono` to declare logic to be executed after the account is deserialized.
- `Single<Account> account` — same as with `Mono` but using RxJava
- `Flux<Account> accounts` — input streaming scenario.

- `Observable<Account> accounts` — input streaming with RxJava.

The response body can be one of the following:

- `Mono<Account>` — serialize without blocking the given Account when the `Mono` completes.
- `Single<Account>` — same but using RxJava.
- `Flux<Account>` — streaming scenario, possibly SSE depending on the requested content type.
- `Observable<Account>` — same but using RxJava `Observable` type.
- `Flowable<Account>` — same but using RxJava 2 `Flowable` type.
- `Publisher<Account>` or `Flow.Publisher<Account>` — any type implementing Reactive Streams `Publisher` is supported.
- `Flux<ServerSentEvent>` — SSE streaming.
- `Mono<Void>` — request handling completes when the `Mono` completes.
- `Account` — serialize without blocking the given Account; implies a synchronous, non-blocking controller method.
- `void` — specific to the annotation-based programming model, request handling completes when the method returns; implies a synchronous, non-blocking controller method.

When using stream types like `Flux` or `Observable`, the media type specified in the request/response or at mapping/routing level is used to determine how the data should be serialized and flushed. For example a REST endpoint that returns a `Flux<Account>` will be serialized by default as following:

- `application/json`: a `Flux<Account>` is handled as an asynchronous collection and serialized as a JSON array with an explicit flush when the `complete` event is emitted.
- `application/stream+json`: a `Flux<Account>` will be handled as a stream of `Account` elements serialized as individual JSON object separated by new lines and explicitly flushed after each element. The `WebClient` supports JSON stream decoding so this is a good use case for server to server use case.
- `text/event-stream`: a `Flux<Account>` or `Flux<ServerSentEvent<Account>>` will be handled as a stream of `Account` or `ServerSentEvent` elements serialized as individual SSE elements using by default JSON for data encoding and explicit flush after each element. This is well suited for exposing a stream to browser clients. `WebClient` supports reading SSE streams as well.

5.3.6. Reactive WebSocket Support

WebFlux includes reactive WebSocket client and server support. Both client and server are supported on the Java WebSocket API (JSR-356), Jetty, Undertow, and Reactor Netty.

On the server side, declare a `WebSocketHandlerAdapter` and then simply add mappings to `WebSocketHandler`-based endpoints:

```
@Bean
public HandlerMapping webSocketMapping() {
    Map<String, WebSocketHandler> map = new HashMap<>();
    map.put("/foo", new FooWebSocketHandler());
    map.put("/bar", new BarWebSocketHandler());

    SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
    mapping.setOrder(10);
    mapping.setUrlMap(map);
    return mapping;
}

@Bean
public WebSocketHandlerAdapter handlerAdapter() {
    return new WebSocketHandlerAdapter();
}
```

On the client side create a `WebSocketClient` for one of the supported libraries listed above:

```
WebSocketClient client = new ReactorNettyWebSocketClient();
client.execute("ws://localhost:8080/echo"), session -> {... }).blockMillis(5000);
```

5.3.7. Testing

The `spring-test` module includes a `WebTestClient` that can be used to test WebFlux server endpoints with or without a running server.

Tests without a running server are comparable to `MockMvc` from Spring MVC where mock request and response are used instead of connecting over the network using a socket. The `WebTestClient` however can also perform tests against a running server.

For more see sample tests in the framework.

5.4. Getting Started

5.4.1. Spring Boot Starter

The Spring Boot WebFlux starter available via http://start.spring.io is the fastest way to get started. It does all that's necessary so you to start writing `@Controller` classes just like with Spring MVC. Simply go to http://start.spring.io, choose version 2.0.0.BUILD-SNAPSHOT, and type reactive in the dependencies box. By default the starter runs with Reactor Netty but the dependencies can be changed as usual with Spring Boot to switch to a different runtime. See the Spring Boot reference documentation page for more details and instruction.

5.4.2. Manual Bootstrapping

This section outlines the steps to get up and running manually.

For dependencies start with `spring-webflux` and `spring-context`. Then add `jackson-databind` and `io.netty:netty-buffer` (temporarily see SPR-14528) for JSON support. Lastly add the dependencies for one of the supported runtimes:

- Tomcat — `org.apache.tomcat.embed:tomcat-embed-core`
- Jetty — `org.eclipse.jetty:jetty-server` and `org.eclipse.jetty:jetty-servlet`
- Reactor Netty — `io.projectreactor.ipc:reactor-netty`
- Undertow — `io.undertow:undertow-core`

For the **annotation-based programming model** bootstrap with:

```
ApplicationContext context = new AnnotationConfigApplicationContext
(DelegatingWebFluxConfiguration.class);  // (1)
HttpHandler handler = DispatcherHandler.toHttpHandler(context);  // (2)
```

The above loads default Spring Web framework configuration (1), then creates a `DispatcherHandler`, the main class driving request processing (2), and adapts it to `HttpHandler` — the lowest level Spring abstraction for reactive HTTP request handling.

For the **functional programming model** bootstrap as follows:

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
// (1)
context.registerBean(FooBean.class, () -> new FooBeanImpl()); // (2)
context.registerBean(BarBean.class); // (3)
context.refresh();

HttpHandler handler = WebHttpHandlerBuilder
        .webHandler(RouterFunctions.toHttpHandler(...))
        .applicationContext(context)
        .build(); // (4)
```

The above creates an `AnnotationConfigApplicationContext` instance (1) that can take advantage of the new functional bean registration API (2) to register beans using a Java 8 `Supplier` or just by specifying its class (3). The `HttpHandler` is created using `WebHttpHandlerBuilder` (4).

The `HttpHandler` can then be installed in one of the supported runtimes:

```
// Tomcat and Jetty (also see notes below)
HttpServlet servlet = new ServletHttpHandlerAdapter(handler);
...

// Reactor Netty
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(handler);
HttpServer.create(host, port).newHandler(adapter).block();

// Undertow
UndertowHttpHandlerAdapter adapter = new UndertowHttpHandlerAdapter(handler);
Undertow server = Undertow.builder().addHttpListener(port, host).setHandler(adapter)
.build();
server.start();
```

For Servlet containers especially with WAR deployment you can use the `AbstractAnnotationConfigDispatcherHandlerInitializer` which as a `WebApplicationInitializer` and is auto-detected by Servlet containers. It takes care of registering the `ServletHttpHandlerAdapter` as shown above. You will need to implement one abstract method in order to point to your Spring configuration.

5.4.3. Examples

You will find code examples useful to build reactive Web application in the following projects:

- Functional programming model sample
- Spring Reactive Playground: playground for most Spring Web reactive features
- Reactor website: the `spring-functional` branch is a Spring 5 functional, Java 8 lambda-style application
- Spring Reactive University session: live-coded project from this Devoxx BE 2106 university talk
- Reactive Thymeleaf Sandbox
- Mix-it 2017 website: Kotlin + Reactive + Functional web and bean registration API application
- Reactor by example: code snippets coming from this InfoQ article
- Spring integration tests: various features tested with Reactor `StepVerifier`

Chapter 6. Spring WebFlux Functional Programming Model

6.1. HandlerFunctions

Incoming HTTP requests are handled by a `HandlerFunction`, which is essentially a function that takes a `ServerRequest` and returns a `Mono<ServerResponse>`. The annotation counterpart to a handler function would be a method with `@RequestMapping`.

`ServerRequest` and `ServerResponse` are immutable interfaces that offer JDK-8 friendly access to the underlying HTTP messages. Both are fully reactive by building on top of Reactor: the request expose the body as `Flux` or `Mono`; the response accepts any Reactive Streams `Publisher` as body.

`ServerRequest` gives access to various HTTP request elements: the method, URI, query parameters, and — through the separate `ServerRequest.Headers` interface — the headers. Access to the body is provided through the `body` methods. For instance, this is how to extract the request body into a `Mono<String>`:

```
Mono<String> string = request.bodyToMono(String.class);
```

And here is how to extract the body into a `Flux`, where `Person` is a class that can be deserialised from the contents of the body (i.e. `Person` is supported by Jackson if the body contains JSON, or JAXB if XML).

```
Flux<Person> people = request.bodyToFlux(Person.class);
```

The two methods above (`bodyToMono` and `bodyToFlux`) are, in fact, convenience methods that use the generic `ServerRequest.body(BodyExtractor)` method. `BodyExtractor` is a functional strategy interface that allows you to write your own extraction logic, but common `BodyExtractor` instances can be found in the `BodyExtractors` utility class. So, the above examples can be replaced with:

```
Mono<String> string = request.body(BodyExtractors.toMono(String.class);
Flux<Person> people = request.body(BodyExtractors.toFlux(Person.class);
```

Similarly, `ServerResponse` provides access to the HTTP response. Since it is immutable, you create a `ServerResponse` with a builder. The builder allows you to set the response status, add response headers, and provide a body. For instance, this is how to create a response with a 200 OK status, a JSON content-type, and a body:

```
Mono<Person> person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person);
```

And here is how to build a response with a 201 Created status, Location header, and empty body:

```
URI location = ...
ServerResponse.created(location).build();
```

Putting these together allows us to create a `HandlerFunction`. For instance, here is an example of a simple "Hello World" handler lambda, that returns a response with a 200 status and a body based on a String:

```
HandlerFunction<ServerResponse> helloWorld =
  request -> ServerResponse.ok().body(fromObject("Hello World"));
```

Writing handler functions as lambda's, as we do above, is convenient, but perhaps lacks in readability and becomes less maintainable when dealing with multiple functions. Therefore, it is recommended to group related handler functions into a handler or controller class. For example, here is a class that exposes a reactive `Person` repository:

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.fromObject;

public class PersonHandler {

    private final PersonRepository repository;

    public PersonHandler(PersonRepository repository) {
        this.repository = repository;
    }

    public Mono<ServerResponse> listPeople(ServerRequest request) { ①
        Flux<Person> people = repository.allPeople();
        return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person
.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) { ②
        Mono<Person> person = request.bodyToMono(Person.class);
        return ServerResponse.ok().build(repository.savePerson(person));
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) { ③
        int personId = Integer.valueOf(request.pathVariable("id"));
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();
        Mono<Person> personMono = this.repository.getPerson(personId);
        return personMono
                .flatMap(person -> ServerResponse.ok().contentType(APPLICATION_JSON)
.body(fromObject(person)))
                .switchIfEmpty(notFound);
    }
}
```

① `listPeople` is a handler function that returns all `Person` objects found in the repository as JSON.

② `createPerson` is a handler function that stores a new `Person` contained in the request body. Note that `PersonRepository.savePerson(Person)` returns `Mono<Void>`: an empty Mono that emits a completion signal when the person has been read from the request and stored. So we use the `build(Publisher<Void>)` method to send a response when that completion signal is received, i.e. when the `Person` has been saved.

③ `getPerson` is a handler function that returns a single person, identified via the path variable `id`. We retrieve that `Person` via the repository, and create a JSON response if it is found. If it is not found, we use `switchIfEmpty(Mono<T>)` to return a 404 Not Found response.

6.2. RouterFunctions

Incoming requests are routed to handler functions with a **`RouterFunction`**, which is a function that takes a `ServerRequest`, and returns a `Mono<HandlerFunction>`. If a request matches a particular route, a handler function is returned; otherwise it returns an empty `Mono`. The `RouterFunction` has a similar purpose as the `@RequestMapping` annotation in `@Controller` classes.

Typically, you do not write router functions yourself, but rather use `RouterFunctions.route(RequestPredicate, HandlerFunction)` to create one using a request predicate and handler function. If the predicate applies, the request is routed to the given handler function; otherwise no routing is performed, resulting in a 404 Not Found response. Though you can write your own `RequestPredicate`, you do not have to: the `RequestPredicates` utility class offers commonly used predicates, such matching based on path, HTTP method, content-type, etc. Using `route`, we can route to our "Hello World" handler function:

```
RouterFunction<ServerResponse> helloWorldRoute =
    RouterFunctions.route(RequestPredicates.path("/hello-world"),
    request -> Response.ok().body(fromObject("Hello World")));
```

Two router functions can be composed into a new router function that routes to either handler function: if the predicate of the first route does not match, the second is evaluated. Composed router functions are evaluated in order, so it makes sense to put specific functions before generic ones. You can compose two router functions by calling `RouterFunction.and(RouterFunction)`, or by calling `RouterFunction.andRoute(RequestPredicate, HandlerFunction)`, which is a convenient combination of `RouterFunction.and()` with `RouterFunctions.route()`.

Given the `PersonHandler` we showed above, we can now define a router function that routes to the respective handler functions. We use method-references to refer to the handler functions:

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> personRoute =
    route(GET("/person/{id}").and(accept(APPLICATION_JSON)), handler::getPerson)
        .andRoute(GET("/person").and(accept(APPLICATION_JSON)), handler::listPeople)
        .andRoute(POST("/person").and(contentType(APPLICATION_JSON)), handler:
:createPerson);
```

Besides router functions, you can also compose request predicates, by calling `RequestPredicate.and(RequestPredicate)` or `RequestPredicate.or(RequestPredicate)`. These work as expected: for `and` the resulting predicate matches if **both** given predicates match; `or` matches if **either** predicate does. Most of the predicates found in `RequestPredicates` are compositions. For instance, `RequestPredicates.GET(String)` is a composition of `RequestPredicates.method(HttpMethod)` and `RequestPredicates.path(String)`.

6.2.1. Running a Server

Now there is just one piece of the puzzle missing: running a router function in an HTTP server. You can convert a router function into a `HttpHandler` by using `RouterFunctions.toHttpHandler(RouterFunction)`. The `HttpHandler` allows you to run on a wide variety of reactive runtimes: Reactor Netty, Servlet 3.1+, and Undertow. Here is how we run a router function in Reactor Netty, for instance:

```
RouterFunction<ServerResponse> route = ...
HttpHandler httpHandler = RouterFunctions.toHttpHandler(route);
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(httpHandler);
HttpServer server = HttpServer.create(HOST, PORT);
server.newHandler(adapter).block();
```

For Tomcat it looks like this:

```
RouterFunction<ServerResponse> route = ...
HttpHandler httpHandler = RouterFunctions.toHttpHandler(route);
HttpServlet servlet = new ServletHttpHandlerAdapter(httpHandler);
Tomcat server = new Tomcat();
Context rootContext = server.addContext("", System.getProperty("java.io.tmpdir"));
Tomcat.addServlet(rootContext, "servlet", servlet);
rootContext.addServletMapping("/", "servlet");
tomcatServer.start();
```

6.3. HandlerFilterFunction

Routes mapped by a router function can be filtered by calling `RouterFunction.filter(HandlerFilterFunction)`, where `HandlerFilterFunction` is essentially a function that takes a `ServerRequest` and `HandlerFunction`, and returns a `ServerResponse`. The handler function parameter represents the next element in the chain: this is typically the `HandlerFunction` that is routed to, but can also be another `FilterFunction` if multiple filters are applied. With annotations, similar functionality can be achieved using `@ControllerAdvice` and/or a `ServletFilter`. Let's add a simple security filter to our route, assuming that we have a `SecurityManager` that can determine whether a particular path is allowed:

```
import static org.springframework.http.HttpStatus.UNAUTHORIZED;

SecurityManager securityManager = ...
RouterFunction<ServerResponse> route = ...

RouterFunction<ServerResponse> filteredRoute =
    route.filter(request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
  });
```

You can see in this example that invoking the `next.handle(ServerRequest)` is optional: we only allow the handler function to be executed when access is allowed.

Chapter 7. Integrating with other web frameworks

7.1. Introduction

This chapter details Spring's integration with third party web frameworks, such as JSF.

One of the core value propositions of the Spring Framework is that of enabling *choice*. In a general sense, Spring does not force one to use or buy into any particular architecture, technology, or methodology (although it certainly recommends some over others). This freedom to pick and choose the architecture, technology, or methodology that is most relevant to a developer and their development team is arguably most evident in the web area, where Spring provides its own web framework (Spring MVC), while at the same time providing integration with a number of popular third party web frameworks. This allows one to continue to leverage any and all of the skills one may have acquired in a particular web framework such as JSF, while at the same time being able to enjoy the benefits afforded by Spring in other areas such as data access, declarative transaction management, and flexible configuration and application assembly.

Having dispensed with the woolly sales patter (c.f. the previous paragraph), the remainder of this chapter will concentrate upon the meaty details of integrating your favorite web framework with Spring. One thing that is often commented upon by developers coming to Java from other languages is the seeming super-abundance of web frameworks available in Java. There are indeed a great number of web frameworks in the Java space; in fact there are far too many to cover with any semblance of detail in a single chapter. This chapter thus picks four of the more popular web frameworks in Java, starting with the Spring configuration that is common to all of the supported web frameworks, and then detailing the specific integration options for each supported web framework.

Please note that this chapter does not attempt to explain how to use any of the supported web frameworks. For example, if you want to use JSF for the presentation layer of your web application, the assumption is that you are already familiar with JSF itself. If you need further details about any of the supported web frameworks themselves, please do consult Further Resources at the end of this chapter.

7.2. Common configuration

Before diving into the integration specifics of each supported web framework, let us first take a look at the Spring configuration that is *not* specific to any one web framework. (This section is equally applicable to Spring's own web framework, Spring MVC.)

One of the concepts (for want of a better word) espoused by (Spring's) lightweight application model is that of a layered architecture. Remember that in a 'classic' layered architecture, the web layer is but one of many layers; it serves as one of the entry points into a server side application and it delegates to service objects (facades) defined in a service layer to satisfy business specific (and presentation-technology agnostic) use cases. In Spring, these service objects, any other

business-specific objects, data access objects, etc. exist in a distinct 'business context', which contains *no* web or presentation layer objects (presentation objects such as Spring MVC controllers are typically configured in a distinct 'presentation context'). This section details how one configures a Spring container (a `WebApplicationContext`) that contains all of the 'business beans' in one's application.

On to specifics: all that one need do is to declare a `ContextLoaderListener` in the standard Java EE servlet `web.xml` file of one's web application, and add a `contextConfigLocation`<context-param/> section (in the same file) that defines which set of Spring XML configuration files to load.

Find below the <listener/> configuration:

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

Find below the <context-param/> configuration:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

If you don't specify the `contextConfigLocation` context parameter, the `ContextLoaderListener` will look for a file called `/WEB-INF/applicationContext.xml` to load. Once the context files are loaded, Spring creates a `WebApplicationContext` object based on the bean definitions and stores it in the `ServletContext` of the web application.

All Java web frameworks are built on top of the Servlet API, and so one can use the following code snippet to get access to this 'business context' `ApplicationContext` created by the `ContextLoaderListener`.

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext
(servletContext);
```

The `WebApplicationContextUtils` class is for convenience, so you don't have to remember the name of the `ServletContext` attribute. Its *getWebApplicationContext()* method will return `null` if an object doesn't exist under the `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE` key. Rather than risk getting `NullPointerExceptions` in your application, it's better to use the `getRequiredWebApplicationContext()` method. This method throws an exception when the `ApplicationContext` is missing.

Once you have a reference to the `WebApplicationContext`, you can retrieve beans by their name or type. Most developers retrieve beans by name and then cast them to one of their implemented interfaces.

Fortunately, most of the frameworks in this section have simpler ways of looking up beans. Not only do they make it easy to get beans from a Spring container, but they also allow you to use dependency injection on their controllers. Each web framework section has more detail on its specific integration strategies.

7.3. JavaServer Faces 1.2

JavaServer Faces (JSF) is the JCP's standard component-based, event-driven web user interface framework. As of Java EE 5, it is an official part of the Java EE umbrella.

For a popular JSF runtime as well as for popular JSF component libraries, check out the Apache MyFaces project. The MyFaces project also provides common JSF extensions such as MyFaces Orchestra: a Spring-based JSF extension that provides rich conversation scope support.

Spring Web Flow 2.0 provides rich JSF support through its newly established Spring Faces module, both for JSF-centric usage (as described in this section) and for Spring-centric usage (using JSF views within a Spring MVC dispatcher). Check out the Spring Web Flow website for details!

The key element in Spring's JSF integration is the JSF `ELResolver` mechanism.

7.3.1. SpringBeanFacesELResolver (JSF 1.2+)

`SpringBeanFacesELResolver` is a JSF 1.2 compliant `ELResolver` implementation, integrating with the standard Unified EL as used by JSF 1.2 and JSP 2.1. Like `SpringBeanVariableResolver`, it delegates to the Spring's 'business context' `WebApplicationContext` *first*, then to the default resolver of the underlying JSF implementation.

Configuration-wise, simply define `SpringBeanFacesELResolver` in your JSF 1.2 *faces-context.xml* file:

```
<faces-config>
    <application>
        <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-
resolver>
        ...
    </application>
</faces-config>
```

7.3.2. FacesContextUtils

A custom `VariableResolver` works well when mapping one's properties to beans in *faces-config.xml*, but at times one may need to grab a bean explicitly. The `FacesContextUtils` class makes this easy. It is similar to `WebApplicationContextUtils`, except that it takes a `FacesContext` parameter rather than a `ServletContext` parameter.

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext
.getCurrentInstance());
```

7.4. Apache Struts 2.x

Invented by Craig McClanahan, Struts is an open source project hosted by the Apache Software Foundation. At the time, it greatly simplified the JSP/Servlet programming paradigm and won over many developers who were using proprietary frameworks. It simplified the programming model, it was open source (and thus free as in beer), and it had a large community, which allowed the project to grow and become popular among Java web developers.

Check out the Struts Spring Plugin for the built-in Spring integration shipped with Struts.

7.5. Tapestry 5.x

From the Tapestry homepage:

Tapestry is a "*Component oriented framework for creating dynamic, robust, highly scalable web applications in Java.*"

While Spring has its own powerful web layer, there are a number of unique advantages to building an enterprise Java application using a combination of Tapestry for the web user interface and the Spring container for the lower layers.

For more information, check out Tapestry's dedicated integration module for Spring.

7.6. Further Resources

Find below links to further resources about the various web frameworks described in this chapter.

- The JSF homepage
- The Struts homepage
- The Tapestry homepage