# Web on Servlet Stack

Version 5.0.3.RELEASE

This part of the documentation covers support for Servlet stack, web applications built on the Servlet API and deployed to Servlet containers. Individual chapters include [Spring MVC](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc), [View Technologies](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-view), [CORS Support](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-cors), and [WebSocket Support](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket). For reactive stack, web applications, go to [Web on Reactive Stack](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#spring-web-reactive).

## 1. Spring Web MVC

### 1.1. Introduction

Spring Web MVC is the original web framework built on the Servlet API and included in the Spring Framework from the very beginning. The formal name "Spring Web MVC" comes from the name of its source module [spring-webmvc](https://github.com/spring-projects/spring-framework/tree/master/spring-webmvc) but it is more commonly known as "Spring MVC".

Parallel to Spring Web MVC, Spring Framework 5.0 introduced a reactive stack, web framework whose name Spring WebFlux is also based on its source module [spring-webflux](https://github.com/spring-projects/spring-framework/tree/master/spring-webflux). This section covers Spring Web MVC. The [next section](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#spring-web-reactive) covers Spring WebFlux.

For baseline information and compatibility with Servlet container and Java EE version ranges please visit the Spring Framework[Wiki](https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-Versions).

### 1.2. DispatcherServlet

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-dispatcher-handler)

Spring MVC, like many other web frameworks, is designed around the front controller pattern where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing while actual work is performed by configurable, delegate components. This model is flexible and supports diverse workflows.

The DispatcherServlet, as any Servlet, needs to be declared and mapped according to the Servlet specification using Java configuration or in web.xml. In turn the DispatcherServlet uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling, [and more](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-servlet-special-bean-types).

Below is an example of the Java configuration that registers and initializes the DispatcherServlet. This class is auto-detected by the Servlet container (see [Container Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-container-config)):

**public** **class** **MyWebApplicationInitializer** **implements** WebApplicationInitializer {

@Override

**public** **void** onStartup(ServletContext servletCxt) {

*// Load Spring web application configuration*

AnnotationConfigWebApplicationContext ac = **new** AnnotationConfigWebApplicationContext();

ac.register(AppConfig.class);

ac.refresh();

*// Create and register the DispatcherServlet*

DispatcherServlet servlet = **new** DispatcherServlet(ac);

ServletRegistration.Dynamic registration = servletCxt.addServlet("app", servlet);

registration.setLoadOnStartup(1);

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

}

}

|  |  |
| --- | --- |
|  | In addition to using the ServletContext API directly, you can also extendAbstractAnnotationConfigDispatcherServletInitializer and override specific methods (see example under [Context Hierarchy](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-servlet-context-hierarchy)). |

Below is an example of web.xml configuration to register and initialize the DispatcherServlet:

<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>

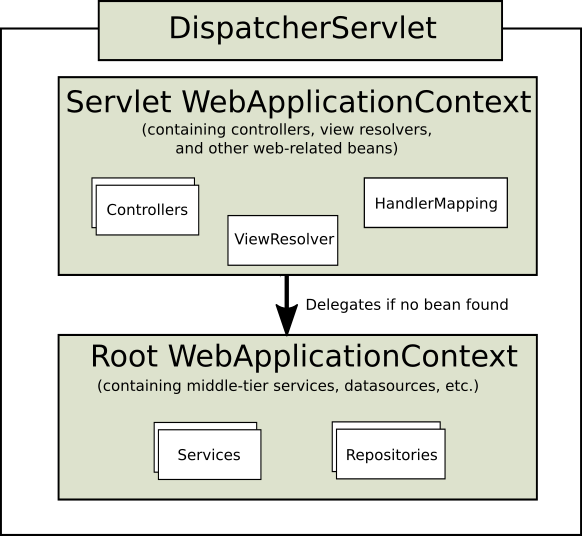
|  |  |
| --- | --- |
|  | Spring Boot follows a different initialization sequence. Rather than hooking into the lifecycle of the Servlet container, Spring Boot uses Spring configuration to bootstrap itself and the embedded Servlet container. Filter and Servlet declarations are detected in Spring configuration and registered with the Servlet container. For more details check the [Spring Boot docs](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-container). |

#### 1.2.1. Context Hierarchy

DispatcherServlet expects a WebApplicationContext, an extension of a plain ApplicationContext, for its own configuration. WebApplicationContext has a link to the ServletContext and Servlet it is associated with. It is also bound to the ServletContext such that applications can use static methods on RequestContextUtils to look up the WebApplicationContextif they need access to it.

For many applications having a single WebApplicationContext is simple and sufficient. It is also possible to have a context hierarchy where one root WebApplicationContext is shared across multiple DispatcherServlet (or other Servlet) instances, each with its own child WebApplicationContext configuration. See [Additional Capabilities of the ApplicationContext](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#context-introduction) for more on the context hierarchy feature.

The root WebApplicationContext typically contains infrastructure beans such as data repositories and business services that need to be shared across multiple Servlet instances. Those beans are effectively inherited and could be overridden (i.e. re-declared) in the Servlet-specific, child WebApplicationContext which typically contains beans local to the given Servlet:



Below is example configuration with 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/\*" };

}

}

|  |  |
| --- | --- |
|  | If an application context hierarchy is not required, applications may return all configuration via getRootConfigClasses() and null from getServletConfigClasses(). |

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>

|  |  |
| --- | --- |
|  | If an application context hierarchy is not required, applications may configure a "root" context only and leave the contextConfigLocation Servlet parameter empty. |

#### 1.2.2. Special Bean Types

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-special-bean-types)

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 one of the framework contracts listed in the table below. Spring MVC provides built-in implementations of these contracts but you can also customize, extend, or replace them.

| **Bean type** | **Explanation** |
| --- | --- |
| [HandlerMapping](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-handlermapping) | Map a request to a handler along with a list of [interceptors](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-handlermapping-interceptor) for pre- and post-processing. The mapping is based on some criteria the details of which vary by HandlerMapping implementation.  The two main HandlerMapping implementations are RequestMappingHandlerMapping which supports @RequestMapping annotated methods and SimpleUrlHandlerMapping which maintains explicit registrations of URI path patterns to handlers. |
| HandlerAdapter | Help 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 annotations. The main purpose of a HandlerAdapteris to shield the DispatcherServlet from such details. |
| [HandlerExceptionResolver](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-exceptionhandlers) | Strategy to resolve exceptions possibly mapping them to handlers, or to HTML error views, or other. See [Exception Handling](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-exceptionhandlers). |
| [ViewResolver](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-viewresolver) | Resolve logical String-based view names returned from a handler to an actual View to render to the response with. See [View Resolution](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-viewresolver) and [View Technologies](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-view). |
| [LocaleResolver](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-localeresolver), [LocaleContextResolver](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-timezone) | Resolve the Locale a client is using and possibly their time zone, in order to be able to offer internationalized views. See [Locale](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-localeresolver). |
| [ThemeResolver](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-themeresolver) | Resolve themes your web application can use, for example, to offer personalized layouts. See [Themes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-themeresolver). |
| [MultipartResolver](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-multipart) | Abstraction for parsing a multi-part request (e.g. browser form file upload) with the help of some multipart parsing library. See [Multipart resolver](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-multipart). |
| [FlashMapManager](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-flash-attributes) | Store and retrieve the "input" and the "output" FlashMap that can be used to pass attributes from one request to another, usually across a redirect. See [Flash attributes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-flash-attributes). |

#### 1.2.3. Framework Config

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-framework-config)

For each type of special bean, the DispatcherServlet checks for the WebApplicationContext first. If there are no matching bean types, it falls back on the default types listed in [DispatcherServlet.properties](https://github.com/spring-projects/spring-framework/blob/master/spring-webmvc/src/main/resources/org/springframework/web/servlet/DispatcherServlet.properties).

Applications can declare the special beans they wish to have. Most applications however will find a better starting point in the MVC Java config or the MVC XML namespace which provide a higher level configuration API that in turn make the necessary bean declarations. See [MVC Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config) for more details.

|  |  |
| --- | --- |
|  | Spring Boot relies on the MVC Java config to configure Spring MVC and also provides many extra convenient options on top. |

#### 1.2.4. Container Config

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 WebApplicationInitializernamed 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.

#### 1.2.5. Processing

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-dispatcher-handler-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 resolver](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-multipart) 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 [Exception Handling](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-exceptionhandlers) 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 1. 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. |

#### 1.2.6. Interception

All HandlerMapping implementations supports handler interceptors that are useful when you want to apply specific functionality to certain requests, for example, checking for a principal. Interceptors must implement HandlerInterceptor from theorg.springframework .web .servlet package with three methods that should provide enough flexibility to do all kinds of pre-processing and post-processing:

* preHandle(..) — before the actual handler is executed
* postHandle(..) — after the handler is executed
* afterCompletion(..) — after the complete request has finished

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.

See [Interceptors](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-interceptors) in the section on MVC configuration for examples of how to configure interceptors. You can also register them directly via setters on individual HandlerMapping implementations.

Note that postHandle is less useful with @ResponseBody and ResponseEntity methods for which a the response is written and committed within the HandlerAdapter and before postHandle. That means its too late to make any changes to the response such as adding an extra header. For such scenarios you can implement ResponseBodyAdvice and either declare it as an [Controller Advice](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-controller-advice) bean or configure it directly on RequestMappingHandlerAdapter.

#### 1.2.7. View Resolution

Spring MVC defines the ViewResolver and View interfaces that enable you to render models in a browser without tying you to a specific view technology. ViewResolver provides a mapping between view names and actual views. View addresses the preparation of data before handing over to a specific view technology.

This table below provides more details on the ViewResolver hierarchy:

| *Table 2. ViewResolver implementations* | |
| --- | --- |
| **ViewResolver** | **Description** |
| AbstractCachingViewResolver | Sub-classes 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. |
| 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, 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 chapter on [View Technologies](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-view). |
| UrlBasedViewResolver | Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings. |
| InternalResourceViewResolver | Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..). See the UrlBasedViewResolverjavadocs 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 [Content negotiation](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-multiple-representations). |

You chain view resolvers by configuring more than one resolver 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.

The contract of a view resolver specifies that a view resolver can return null to indicate the view could not be found. However in the case of JSPs, and InternalResourceViewResolver, the only way to figure out if a JSP exists is to perform a dispatch through RequestDispatcher. There an InternalResourceViewResolver must always be configured last in the order.

See [View Controllers](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-view-controller) under [MVC Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config) for details on how to configure view resolution. Also see[View Technologies](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-view) for more details on supported view technologies.

##### Redirect

The special redirect: prefix in a view name allows you to perform a redirect. The UrlBasedViewResolver (and 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 if a controller method is annotated with the @ResponseStatus, the annotation value takes precedence over the response status set by RedirectView.

##### Forward

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 does a RequestDispatcher.forward(). Therefore, this prefix is not useful with InternalResourceViewResolver and InternalResourceView (for JSPs) but it can be helpful if 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.

##### Content negotiation

[ContentNegotiatingViewResolver](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/view/ContentNegotiatingViewResolver.html) does not resolve views itself but rather delegates to other view resolvers, and selects the view that resembles the representation requested by the client. The representation can be determined from the Accept header or from a query parameter, e.g. "/path?format=pdf".

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.

See [View Resolvers](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-view-resolvers) under [MVC Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config) for configuration details.

#### 1.2.8. Locale

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 LocaleResolverobjects.

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 [Interception](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-handlermapping-interceptor) for more information on handler mapping interceptors) to change the locale under specific circumstances, for example, based on a parameter in the request.

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

##### TimeZone

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.

##### Header resolver

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.

##### Cookie resolver

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 3. 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. |

##### Session resolver

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.

##### Locale interceptor

You can enable changing of locales by adding the LocaleChangeInterceptor to one of the handler mappings (see [[mvc-handlermapping]](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-handlermapping)). It will detect a parameter in the request and change the locale. It calls setLocale() on the LocaleResolverthat 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](https://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>

#### 1.2.9. 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.

##### Define a theme

To use themes in your web application, you must set up an implementation of theorg.springframework.ui.context.ThemeSource interface. The WebApplicationContext interface extends ThemeSource but delegates its responsibilities to a dedicated implementation. By default the delegate will be anorg.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.

##### Resolve themes

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 4. 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.

#### 1.2.10. Multipart resolver

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-multipart)

MultipartResolver from the org.springframework.web.multipart package is a strategy for parsing multipart requests including file uploads. There is one implementation based on [Commons FileUpload](https://jakarta.apache.org/commons/fileupload) and another based on Servlet 3.0 multipart request parsing.

To enable multipart handling, you need declare a MultipartResolver bean in your DispatcherServlet Spring configuration with the name "multipartResolver". The DispatcherServlet detects it and applies it to incoming request. When a POST with content-type of "multipart/form-data" is received, the resolver parses the content and wraps the current HttpServletRequest as MultipartHttpServletRequest in order to provide access to resolved parts in addition to exposing them as request parameters.

##### Apache FileUpload

To use Apache Commons FileUpload, simply configure a bean of type CommonsMultipartResolver with the name multipartResolver. Of course you also need to have commons-fileupload as a dependency on your classpath.

##### Servlet 3.0

To use Servlet 3.0 multipart support, you need to register the DispatcherServlet accordingly. In programmatic Servlet registration, set a MultipartConfigElement on the Servlet registration. In web.xml, add a "<multipart-config>" section. Configuration settings such as maximum sizes or storage locations need to be applied at this level since Servlet 3.0 API does not make it possible for the MultipartResolver to do so.

Once the Servlet 3.0 configuration is in place, simply add a bean of type StandardServletMultipartResolver with the name multipartResolver.

### 1.3. Filters

The spring-web module provides some useful filters.

#### 1.3.1. HTTP PUT Form

Browsers can only submit form data via HTTP GET or HTTP POST but non-browser clients can also use HTTP PUT and PATCH. The Servlet API requires ServletRequest.getParameter\*() methods to support form field access only for HTTP POST.

The spring-web module provides HttpPutFormContentFilter that 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.

#### 1.3.2. 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](https://tools.ietf.org/html/rfc7239) 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".

ForwardedHeaderFilter detects, extracts, and uses information from the "Forwarded" header, or from "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto". It wraps the request in order to overlay its host, port, and scheme and also "hides" the 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.

#### 1.3.3. Shallow ETag

There is a ShallowEtagHeaderFilter. It is called shallow because it doesn’t have any knowledge of the content. Instead it relies on buffering actual content written to the response and computing the ETag value at the end.

See [ETag Filter](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-httpcaching-shallowetag) for more details.

#### 1.3.4. CORS

Spring MVC provides fine-grained support for CORS configuration through annotations on controllers. However when used with Spring Security it is advisable to rely on the built-in CorsFilter that must be ordered ahead of Spring Security’s chain of filters.

See the section on [CORS](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-cors) and the [CorsFilter](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-cors-filter) for more details.

### 1.4. Annotated Controllers

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-controller)

Spring MVC provides an annotation-based programming model where @Controller and @RestController components use annotations to express request mappings, request input, exception handling, and more. Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces.

@Controller

**public** **class** **HelloController** {

@GetMapping("/hello")

**public** String handle(Model model) {

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

**return** "index";

}

}

In this particular example the method accepts a Model and returns a view name as a String but many other options exist and are explained further below in this chapter.

|  |  |
| --- | --- |
|  | Guides and tutorials on [spring.io](https://spring.io/guides) use the annotation-based programming model described in this section. |

#### 1.4.1. Declaration

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-controller)

You can define controller beans using a standard Spring bean definition in the Servlet’s WebApplicationContext. The @Controller stereotype allows for auto-detection, aligned with Spring general support for detecting @Component classes in the classpath and auto-registering bean definitions for them. It also acts as a stereotype for the annotated class, indicating its role as a web component.

To enable auto-detection of such @Controller beans, you can add component scanning to your Java configuration:

@Configuration

@ComponentScan("org.example.web")

**public** **class** **WebConfig** {

*// ...*

}

The XML configuration equivalent:

<?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.example.web"/>

*<!-- ... -->*

</beans>

@RestController is a composed annotation that is itself annotated with @Controller and @ResponseBody indicating a controller whose every method inherits the type-level @ResponseBody annotation and therefore writes to the response body (vs model-and-vew rendering).

##### AOP proxies

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"/>.

#### 1.4.2. Request Mapping

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestmapping)

The @RequestMapping annotation is used to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. It can be used at the class-level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

There are also HTTP method specific shortcut variants of @RequestMapping:

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

The shortcut variants are [composed annotations](https://github.com/spring-projects/spring-framework/wiki/Spring-Annotation-Programming-Model#composed-annotations) — themselves annotated with @RequestMapping. They are commonly used at the method level. At the class level an @RequestMapping is more useful for expressing shared mappings.

@RestController

@RequestMapping("/persons")

**class** **PersonController** {

@GetMapping("/{id}")

**public** Person getPerson(@PathVariable Long id) {

*// ...*

}

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

**public** **void** add(@RequestBody Person person) {

*// ...*

}

}

##### URI patterns

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestmapping-uri-templates)

You can map requests using glob patterns and wildcards:

* ? matches one character
* \* matches zero or more characters within a path segment
* \*\* match zero or more path segments

You can also declare URI variables and access their values with @PathVariable:

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

**public** Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

*// ...*

}

URI variables can be declared at the class and method level:

@Controller

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

**public** **class** **OwnerController** {

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

**public** Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

*// ...*

}

}

URI variables are automatically converted to the appropriate type or`TypeMismatchException` is raised. Simple types — int, long, Date, are supported by default and you can register support for any other data type. See [Type Conversion](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-typeconversion) and [Binder Methods](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-initbinder).

URI variables can be named explicitly — e.g. @PathVariable("customId"), but you can leave that detail out if the names are the same and your code is compiled with debugging information or with the -parameters compiler flag on Java 8.

The syntax {varName:regex} declares a URI variable with a regular expressions with the syntax {varName:regex} — e.g. given URL "/spring-web-3.0.5 .jar", the below method extracts the name, version, and file extension:

@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")

**public** **void** handle(@PathVariable String version, @PathVariable String ext) {

*// ...*

}

URI path patterns can also have embedded ${…​} placeholders that are resolved on startup via PropertyPlaceHolderConfigureragainst local, system, environment, and other property sources. This can be used for example to parameterize a base URL based on some external configuration.

|  |  |
| --- | --- |
|  | Spring MVC uses the PathMatcher contract and the AntPathMatcher implementation from spring-core for URI path matching. |

##### Pattern comparison

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestmapping-pattern-comparison)

When multiple patterns match a URL, they must be compared to find the best match. This done via AntPathMatcher.getPatternComparator(String path) which looks for patterns that more specific.

A pattern is less specific if it has a lower count of URI variables and single wildcards counted as 1 and double wildcards counted as 2. Given an equal score, the longer pattern is chosen. Given the same score and length, the pattern with more URI variables than wildcards is chosen.

The default mapping pattern /\*\* is excluded from scoring and always sorted last. Also prefix patterns such as /public/\*\* are considered less specific than other pattern that don’t have double wildcards.

For the full details see AntPatternComparator in AntPathMatcher and also keep mind that the PathMatcher implementation used can be customized. See [Path Matching](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-path-matching) in the configuration section.

##### Suffix match

By default Spring MVC performs ".\*" suffix pattern matching so that a controller mapped to /person is also implicitly mapped to /person.\*. The file extension is then used to interpret the requested content type to use for the response (i.e. instead of the "Accept" header), e.g. /person.pdf, /person.xml, etc.

Using file extensions like this was necessary when browsers used to send Accept headers that were hard to interpret consistently. At present that is no longer a necessity and using the "Accept" header should be the preferred choice.

Over time the use of file name extensions has proven problematic in a variety of ways. It can cause ambiguity when overlayed with the use of URI variables, path parameters, URI encoding, and it also makes it difficult to reason about URL-based authorization and security (see next section for more details).

To completely disable the use of file extensions, you must set both of these:

* useSuffixPatternMatching(false), see [PathMatchConfigurer](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-path-matching)
* favorPathExtension(false), see [ContentNeogiationConfigurer](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-content-negotiation)

URL-based content negotiation can still be useful, for example when typing a URL in a browser. To enable that we recommend a query parameter based strategy to avoid most of the issues that come with file extensions. Or if you must use file extensions, consider restricting them to a list of explicitly registered extensions through the mediaTypes property of [ContentNeogiationConfigurer](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-content-negotiation).

##### Suffix match and RFD

Reflected file download (RFD) attack is similar to XSS in that it relies on request 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 when double-clicked later.

In Spring MVC @ResponseBody and ResponseEntity methods are at risk because they can render different content types which clients can request via URL path extensions. Disabling suffix pattern matching and the use of path extensions for content negotiation lower the risk but are not sufficient to prevent RFD attacks.

To prevent RFD attacks, prior to rendering the response body Spring MVC adds a Content-Disposition:inline;filename=f.txtheader 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. Applications with custom HttpMessageConverter implementations can explicitly register file extensions for content negotiation to avoid having a Content-Disposition header added for those extensions. See [Content Types](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-content-negotiation).

Check [CVE-2015-5211](https://pivotal.io/security/cve-2015-5211) for additional recommendations related to RFD.

##### Consumable media types

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestmapping-consumes)

You can narrow the request mapping based on the Content-Type of the request:

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

**public** **void** addPet(@RequestBody Pet pet) {

*// ...*

}

The consumes attribute also supports negation expressions — e.g. !text/plain means any content type other than "text/plain".

You can declare a shared consumes attribute at the class level. Unlike most other request mapping attributes however when used at the class level, a method-level consumes attribute will overrides rather than extend the class level declaration.

|  |  |
| --- | --- |
|  | MediaType provides constants for commonly used media types — e.g. APPLICATION\_JSON\_VALUE, APPLICATION\_JSON\_UTF8\_VALUE. |

##### Producible media types

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestmapping-produces)

You can narrow the request mapping based on the Accept request header and the list of content types that a controller method produces:

@GetMapping(path = "/pets/{petId}", **produces = "application/json;charset=UTF-8"**)

@ResponseBody

**public** Pet getPet(@PathVariable String petId) {

*// ...*

}

The media type can specify a character set. Negated expressions are supported — e.g. !text/plain means any content type other than "text/plain".

You can declare a shared produces attribute at the class level. Unlike most other request mapping attributes however when used at the class level, a method-level produces attribute will overrides rather than extend the class level declaration.

|  |  |
| --- | --- |
|  | MediaType provides constants for commonly used media types — e.g. APPLICATION\_JSON\_VALUE, APPLICATION\_JSON\_UTF8\_VALUE. |

##### Parameters, headers

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestmapping-params-and-headers)

You can narrow request mappings based on request parameter conditions. You can test for the presence of a request parameter ("myParam"), for the absence ("!myParam"), or for a specific value ("myParam=myValue"):

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

**public** **void** findPet(@PathVariable String petId) {

*// ...*

}

You can also use the same with request header conditions:

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

**public** **void** findPet(@PathVariable String petId) {

*// ...*

}

|  |  |
| --- | --- |
|  | You can match Content-Type and Accept with the headers condition but it is better to use [consumes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-consumes) and [produces](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-produces) instead. |

##### HTTP HEAD, OPTIONS

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestmapping-head-options)

@GetMapping — and also @RequestMapping(method=HttpMethod.GET), support HTTP HEAD transparently for request mapping purposes. Controller methods don’t need to change. A response wrapper, applied in javax.servlet.http.HttpServlet, ensures a "Content-Length" header is set to the number of bytes written and without actually writing to the response.

@GetMapping — and also @RequestMapping(method=HttpMethod.GET), are implicitly mapped to and also support HTTP HEAD. An HTTP HEAD request is processed as if it were HTTP GET except but instead of writing the body, the number of bytes are counted and the "Content-Length" header set.

By default HTTP OPTIONS is handled by setting the "Allow" response header to the list of HTTP methods listed in all @RequestMapping methods with matching URL patterns.

For a @RequestMapping without HTTP method declarations, the "Allow" header is set to"GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS". Controller methods should always declare the supported HTTP methods for example by using the HTTP method specific variants — @GetMapping, @PostMapping, etc.

@RequestMapping method can be explicitly mapped to HTTP HEAD and HTTP OPTIONS, but that is not necessary in the common case.

#### 1.4.3. Handler Methods

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-methods)

@RequestMapping handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.

##### Method Arguments

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-arguments)

The table below shows supported controller method arguments. Reactive types are not supported for any arguments.

JDK 8’s java.util.Optional is supported as a method argument in combination with annotations that have a requiredattribute — e.g. @RequestParam, @RequestHeader, etc, and is equivalent to required=false.

| **Controller method argument** | **Description** |
| --- | --- |
| WebRequest, NativeWebRequest | Generic access to request parameters, request & session attributes, without direct use of the Servlet API. |
| javax.servlet.ServletRequest, javax.servlet.ServletResponse | Choose any specific request or response type — e.g. ServletRequest, HttpServletRequest, or Spring’s MultipartRequest, MultipartHttpServletRequest. |
| javax.servlet.http.HttpSession | Enforces the presence of a session. As a consequence, such an argument is never null. **Note:** Session access is not thread-safe. Consider setting theRequestMappingHandlerAdapter's "synchronizeOnSession" flag to "true" if multiple requests are allowed to access a session concurrently. |
| javax.servlet.http.PushBuilder | Servlet 4.0 push builder API for programmatic HTTP/2 resource pushes. Note that per Servlet spec, the injected PushBuilder instance can be null if the client does not support that HTTP/2 feature. |
| java.security.Principal | Currently authenticated user; possibly a specific Principal implementation class if known. |
| HttpMethod | The HTTP method of the request. |
| java.util.Locale | The current request locale, determined by the most specific LocaleResolveravailable, in effect, the configured LocaleResolver/LocaleContextResolver. |
| Java 6+: java.util.TimeZone Java 8+: java.time.ZoneId | The time zone associated with the current request, as determined by a LocaleContextResolver. |
| java.io.InputStream, java.io.Reader | For access to the raw request body as exposed by the Servlet API. |
| java.io.OutputStream, java.io.Writer | For access to the raw response body as exposed by the Servlet API. |
| @PathVariable | For access to URI template variables. See [URI patterns](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-uri-templates). |
| @MatrixVariable | For access to name-value pairs in URI path segments. See [Matrix variables](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-matrix-variables). |
| @RequestParam | For access to Servlet request parameters. Parameter values are converted to the declared method argument type. See [@RequestParam](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestparam).  Note that use of @RequestParam is optional, e.g. to set its attributes. See "Any other argument" further below in this table. |
| @RequestHeader | For access to request headers. Header values are converted to the declared method argument type. See [@RequestHeader](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestheader). |
| @CookieValue | For access to cookies. Cookies values are converted to the declared method argument type. See [@CookieValue](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-cookievalue). |
| @RequestBody | For access to the HTTP request body. Body content is converted to the declared method argument type using HttpMessageConverters. See [@RequestBody](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestbody). |
| HttpEntity<B> | For access to request headers and body. The body is converted with HttpMessageConverters. See [HttpEntity](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-httpentity). |
| @RequestPart | For access to a part in a "multipart/form-data" request. See [Multipart](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-multipart-forms). |
| java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap | For access to the model that is used in HTML controllers and exposed to templates as part of view rendering. |
| RedirectAttributes | Specify attributes to use in case of a redirect — i.e. to be appended to the query string, and/or flash attributes to be stored temporarily until the request after redirect. See [Redirect attributes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-redirecting-passing-data) and [Flash attributes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-flash-attributes). |
| @ModelAttribute | For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See [@ModelAttribute](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-modelattrib-method-args) as well as [Model Methods](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-modelattrib-methods) and [Binder Methods](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-initbinder).  Note that use of @ModelAttribute is optional, e.g. to set its attributes. See "Any other argument" further below in this table. |
| Errors, BindingResult | For access to errors from validation and data binding for a command object (i.e. @ModelAttribute argument), or errors from the validation of an @RequestBody or@RequestPart arguments; an Errors, or BindingResult argument must be declared immediately after the validated method argument. |
| SessionStatus + class-level @SessionAttributes | For marking form processing complete which triggers cleanup of session attributes declared through a class-level @SessionAttributes annotation. See[@SessionAttributes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-sessionattributes) for more details. |
| UriComponentsBuilder | For preparing a URL relative to the current request’s host, port, scheme, context path, and the literal part of the servlet mapping also taking into account Forwarded and X-Forwarded-\* headers. See [URI Links](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-uri-building). |
| @SessionAttribute | For access to any session attribute; in contrast to model attributes stored in the session as a result of a class-level @SessionAttributes declaration. See[@SessionAttribute](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-sessionattribute) for more details. |
| @RequestAttribute | For access to request attributes. See [@RequestAttribute](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestattrib) for more details. |
| Any other argument | If a method argument is not matched to any of the above, by default it is resolved as an @RequestParam if it is a simple type, as determined by[BeanUtils#isSimpleProperty](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/BeanUtils.html#isSimpleProperty-java.lang.Class-), or as an @ModelAttribute otherwise. |

##### Return Values

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-return-types)

The table below shows supported controller method return values. Reactive types are supported for all return values, see below for more details.

| **Controller method return value** | **Description** |
| --- | --- |
| @ResponseBody | The return value is converted through HttpMessageConverters and written to the response. See [@ResponseBody](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-responsebody). |
| HttpEntity<B>, ResponseEntity<B> | The return value specifies the full response including HTTP headers and body be converted through HttpMessageConverters and written to the response. See [ResponseEntity](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-responseentity). |
| HttpHeaders | For returning a response with headers and no body. |
| String | A view name to be resolved with ViewResolver's and used together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above). |
| View | A View instance to use for rendering together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above). |
| java.util.Map, org.springframework.ui.Model | Attributes to be added to the implicit model with the view name implicitly determined through a RequestToViewNameTranslator. |
| @ModelAttribute | An attribute to be added to the model with the view name implicitly determined through a RequestToViewNameTranslator.  Note that @ModelAttribute is optional. See "Any other return value" further below in this table. |
| ModelAndView object | The view and model attributes to use, and optionally a response status. |
| void | A method with a void return type (or null return value) is considered to have fully handled the response if it also has a ServletResponse, or an OutputStream argument, or an @ResponseStatus annotation. The same is true also if the controller has made a positive ETag or lastModified timestamp check (see [@Controller caching](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-caching-etag-lastmodified) for details).  If none of the above is true, a void return type may also indicate "no response body" for REST controllers, or default view name selection for HTML controllers. |
| Callable<V> | Produce any of the above return values asynchronously in a Spring MVC managed thread. |
| DeferredResult<V> | Produce any of the above return values asynchronously from any thread — e.g. possibly as a result of some event or callback. |
| ListenableFuture<V>,java.util.concurrent.CompletionStage<V>,java.util.concurrent.CompletableFuture<V> | Alternative to DeferredResult as a convenience for example when an underlying service returns one of those. |
| ResponseBodyEmitter, SseEmitter | Emit a stream of objects asynchronously to be written to the response withHttpMessageConverter's; also supported as the body of a ResponseEntity. |
| StreamingResponseBody | Write to the response OutputStream asynchronously; also supported as the body of a ResponseEntity. |
| Reactive types — Reactor, RxJava, or others via ReactiveAdapterRegistry | Alternative to `DeferredResult with multi-value streams (e.g. Flux, Observable) collected to a List.  For streaming scenarios — .e.g. text/event-stream, application/json+stream, SseEmitter and ResponseBodyEmitter are used instead, where ServletOutputStream blocking I/O is performed on a Spring MVC managed thread and back pressure applied against the completion of each write.  See [Reactive return values](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-reactive-types). |
| Any other return value | If a return value is not matched to any of the above, by default it is treated as a view name, if it is String or void (default view name selection viaRequestToViewNameTranslator applies); or as a model attribute to be added to the model, unless it is a simple type, as determined by[BeanUtils#isSimpleProperty](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/BeanUtils.html#isSimpleProperty-java.lang.Class-) in which case it remains unresolved. |

##### Type Conversion

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-typeconversion)

Some annotated controller method arguments that represent String-based request input — e.g. @RequestParam, @RequestHeader, @PathVariable, @MatrixVariable, and @CookieValue, may require type conversion if the argument is declared as something other than String.

For such cases type conversion is automatically applied based on the configured converters. By default simple types such as int, long, Date, etc. are supported. Type conversion can be customized through a WebDataBinder, see [Binder Methods](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-initbinder), or by registering Formatters with the FormattingConversionService, see [Spring Field Formatting](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#format).

##### Matrix variables

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-matrix-variables)

[RFC 3986](https://tools.ietf.org/html/rfc3986#section-3.3) discusses name-value pairs in path segments. In Spring MVC we refer to those as "matrix variables" based on an ["old post"](https://www.w3.org/DesignIssues/MatrixURIs.html) by Tim Berners-Lee but they can be also be referred to as URI path parameters.

Matrix variables can appear in any path segment, each variable separated by semicolon and multiple values separated by comma, e.g. "/cars;color=red,green;year=2012". Multiple values can also be specified through repeated variable names, e.g."color=red;color=green;color=blue".

If a URL is expected to contain matrix variables, the request mapping for a controller method must use a URI variable to mask that variable content and ensure the request can be matched successfully independent of matrix variable order and presence. Below is an example:

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

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

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

*// petId == 42*

*// q == 11*

}

Given that all path segments may contain matrix variables, sometimes you may need to disambiguate which path variable the matrix variable is expected to be in. For example:

*// 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*

}

To get all matrix variables, use a MultiValueMap:

*// 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 you need to enable the use of matrix variables. In the MVC Java config you need to set a UrlPathHelper with removeSemicolonContent=false via [Path Matching](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-path-matching). In the MVC XML namespace, use <mvc:annotation-driven enable-matrix-variables="true"/>.

##### @RequestParam

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestparam)

Use the @RequestParam annotation to bind Servlet request parameters (i.e. query parameters or form data) to a method argument in a controller.

The following code snippet shows the usage:

@Controller

@RequestMapping("/pets")

**public** **class** **EditPetForm** {

*// ...*

@GetMapping

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

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

model.addAttribute("pet", pet);

**return** "petForm";

}

*// ...*

}

Method parameters using this annotation are required by default, but you can specify that a method parameter is optional by setting @RequestParam's required flag to false or by declaring the argument with an java.util.Optional wrapper.

Type conversion is applied automatically if the target method parameter type is not String. See [Type Conversion](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-typeconversion).

When an @RequestParam annotation is declared as Map<String, String> or MultiValueMap<String, String> argument, the map is populated with all request parameters.

Note that use of @RequestParam is optional, e.g. to set its attributes. By default any argument that is a simple value type, as determined by [BeanUtils#isSimpleProperty](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/BeanUtils.html#isSimpleProperty-java.lang.Class-), and is not resolved by any other argument resolver, is treated as if it was annotated with @RequestParam.

##### @RequestHeader

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestheader)

Use the @RequestHeader annotation to bind a request header to a method argument in a controller.

Given request with headers:

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 gets the value of the Accept-Encoding and Keep-Alive headers:

@GetMapping("/demo")

**public** **void** handle(

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

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

*//...*

}

Type conversion is applied automatically if the target method parameter type is not String. See [Type Conversion](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-typeconversion).

When an @RequestHeader annotation is used on a Map<String, String>, MultiValueMap<String, String>, or HttpHeadersargument, 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>. |

##### @CookieValue

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-cookievalue)

Use the @CookieValue annotation to bind the value of an HTTP cookie to a method argument in a controller.

Given request with the following cookie:

JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84

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

@GetMapping("/demo")

**public** **void** handle(**@CookieValue("JSESSIONID")** String cookie) {

*//...*

}

Type conversion is applied automatically if the target method parameter type is not String. See [Type Conversion](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-typeconversion).

##### @ModelAttribute

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-modelattrib-method-args)

Use the @ModelAttribute annotation on a method argument to access an attribute from the model, or have it instantiated if not present. The model attribute is also overlaid with values from HTTP Servlet request parameters whose names match to field names. This is referred to as data binding and it saves you from having to deal with parsing and converting individual query parameters and form fields. For example:

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

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

The Pet instance above is resolved as follows:

* From the model if already added via [Model Methods](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-modelattrib-methods).
* From the HTTP session via [@SessionAttributes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-sessionattributes).
* From a URI path variable passed through a Converter (example below).
* From the invocation of a default constructor.
* From the invocation of a "primary constructor" with arguments matching to Servlet request parameters; argument names are determined via JavaBeans @ConstructorProperties or via runtime-retained parameter names in the bytecode.

While it is common to use a [Model Methods](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-modelattrib-methods) to populate the model with attributes, one other alternative is to rely on a Converter<String, T> in combination with a URI path variable convention. In the example below the model attribute name "account" matches the URI path variable "account" and the Account is loaded by passing the String account number through a registered Converter<String, Account>:

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

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

*// ...*

}

After the model attribute instance is obtained, data binding is applied. The WebDataBinder class matches Servlet request parameter names (query parameters and form fields) to field names on the target Object. Matching fields are populated after type conversion is applied where necessary. For more on data binding (and validation) see [Validation](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#validation). For more on customizing data binding see [Binder Methods](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-initbinder).

Data binding may result in errors. By default a BindException is raised but to check for such errors in the controller method, add a BindingResult argument immediately next to the @ModelAttribute as shown below:

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

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

**if** (result.hasErrors()) {

**return** "petForm";

}

*// ...*

}

In some cases you may want access to a model attribute without data binding. For such cases you can inject the Model into the controller and access it directly or alternatively set @ModelAttribute(binding=false) as shown below:

@ModelAttribute

**public** AccountForm setUpForm() {

**return** **new** AccountForm();

}

@ModelAttribute

**public** Account findAccount(@PathVariable String accountId) {

**return** accountRepository.findOne(accountId);

}

@PostMapping("update")

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

**@ModelAttribute(binding=false)** Account account) {

*// ...*

}

Validation can be applied automatically after data binding by adding the javax.validation.Valid annotation or Spring’s @Validated annotation (also see [Bean validation](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#validation-beanvalidation) and [Spring validation](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#validation)). For example:

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

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

**if** (result.hasErrors()) {

**return** "petForm";

}

*// ...*

}

Note that use of @ModelAttribute is optional, e.g. to set its attributes. By default any argument that is not a simple value type, as determined by [BeanUtils#isSimpleProperty](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/BeanUtils.html#isSimpleProperty-java.lang.Class-), and is not resolved by any other argument resolver, is treated as if it was annotated with @ModelAttribute.

##### @SessionAttributes

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-sessionattributes)

@SessionAttributes is used to store model attributes in the HTTP Servlet session between requests. It is a type-level annotation that declares session attributes used by a specific controller. This will typically list the names of model attributes or types of model attributes which should be transparently stored in the session for subsequent requests to access.

For example:

@Controller

**@SessionAttributes("pet")**

**public** **class** **EditPetForm** {

*// ...*

}

On the first request when a model attribute with the name "pet" is added to the model, it is automatically promoted to and saved in the HTTP Servlet session. It remains there until another controller method uses a SessionStatus method argument to clear the storage:

@Controller

**@SessionAttributes("pet")**

**public** **class** **EditPetForm** {

*// ...*

@PostMapping("/pets/{id}")

**public** String handle(Pet pet, BindingResult errors, SessionStatus status) {

**if** (errors.hasErrors) {

*// ...*

}

status.setComplete();

*// ...*

}

}

}

##### @SessionAttribute

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-sessionattribute)

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(**@SessionAttribute** User user) {

*// ...*

}

For use cases that require adding or removing session attributes consider injectingorg.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 [@SessionAttributes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-sessionattributes).

##### @RequestAttribute

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestattrib)

Similar to @SessionAttribute the @RequestAttribute annotation can be used to access pre-existing request attributes created earlier, e.g. by a Servlet Filter or HandlerInterceptor:

@GetMapping("/")

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

*// ...*

}

##### Redirect attributes

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

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 [Flash attributes](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-flash-attributes) for more information.

##### 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 viaRedirectAttributes 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 .

##### Multipart

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-multipart-forms)

After a MultipartResolver has been [enabled](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-multipart), the content of POST requests with "multipart/form-data" is parsed and accessible as regular request parameters. In the example below we access one regular form field and one uploaded file:

@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";

}

}

|  |  |
| --- | --- |
|  | When using Servlet 3.0 multipart parsing you can also use javax.servlet.http.Part as a method argument instead of Spring’s MultipartFile. |

Multipart content can also be used as part of data binding to a [command object](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-modelattrib-method-args). For example the above form field and file could have been fields on a form object:

**class** **MyForm** {

**private** String name;

**private** MultipartFile file;

*// ...*

}

@Controller

**public** **class** **FileUploadController** {

@PostMapping("/form")

**public** String handleFormUpload(MyForm form, BindingResult errors) {

**if** (!form.getFile().isEmpty()) {

**byte[]** bytes = form.getFile().getBytes();

*// store the bytes somewhere*

**return** "redirect:uploadSuccess";

}

**return** "redirect:uploadFailure";

}

}

Multipart requests can also be submitted from non-browser clients in a RESTful service scenario. For example a file along with JSON:

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 can access the "meta-data" part with @RequestParam as a String but you’ll probably want it deserialized from JSON (similar to @RequestBody). Use the @RequestPart annotation to access a multipart after converting it with an [HttpMessageConverter](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/integration.html#rest-message-conversion):

@PostMapping("/")

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

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

*// ...*

}

@RequestPart can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a MethodArgumentNotValidException which is turned into a 400 (BAD\_REQUEST) response. Alternatively validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@PostMapping("/")

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

**BindingResult result**) {

*// ...*

}

##### @RequestBody

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-requestbody)

Use the @RequestBody annotation to have the request body read and deserialized into an Object through an [HttpMessageConverter](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/integration.html#rest-message-conversion). Below is an example with an @RequestBody argument:

@PostMapping("/accounts")

**public** **void** handle(@RequestBody Account account) {

*// ...*

}

You can use the [Message Converters](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-message-converters) option of the [MVC Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config) to configure or customize message conversion.

@RequestBody can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a MethodArgumentNotValidException which is turned into a 400 (BAD\_REQUEST) response. Alternatively validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@PostMapping("/accounts")

**public** **void** handle(@Valid @RequestBody Account account, BindingResult result) {

*// ...*

}

##### HttpEntity

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-httpentity)

HttpEntity is more or less identical to using [@RequestBody](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestbody) but based on a container object that exposes request headers and body. Below is an example:

@PostMapping("/accounts")

**public** **void** handle(HttpEntity<Account> entity) {

*// ...*

}

##### @ResponseBody

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-responsebody)

Use the @ResponseBody annotation on a method to have the return serialized to the response body through an[HttpMessageConverter](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/integration.html#rest-message-conversion). For example:

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

@ResponseBody

**public** Account handle() {

*// ...*

}

@ResponseBody is also supported at the class level in which case it is inherited by all controller methods. This is the effect of @RestController which is nothing more than a meta-annotation marked with @Controller and @ResponseBody.

@ResponseBody may be used with reactive types. See [Async Requests](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async) and [Reactive return values](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-reactive-types) for more details.

You can use the [Message Converters](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-message-converters) option of the [MVC Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config) to configure or customize message conversion.

@ResponseBody methods can be combined with JSON serialization views. See [Jackson JSON](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-jackson) for details.

##### ResponseEntity

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-responseentity)

ResponseEntity is more or less identical to using [@ResponseBody](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-responsebody) but based on a container object that specifies request headers and body. Below is an example:

@PostMapping("/something")

**public** ResponseEntity<String> handle() {

*// ...*

URI location = ...

return **new** ResponseEntity.created(location).build();

}

##### Jackson JSON

###### Jackson serialization views

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-jsonview)

Spring MVC provides built-in support for [Jackson’s Serialization Views](http://wiki.fasterxml.com/JacksonJsonViews) which allows rendering only a subset of all fields in an Object. To use it with @ResponseBody or ResponseEntity controller methods, use Jackson’s @JsonView annotation to activate a serialization view class:

@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;

}

}

|  |  |
| --- | --- |
|  | @JsonView allows an array of view classes but you can only specify only one per controller method. Use a composite interface if you need to activate 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

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

@ControllerAdvice

**public** **class** **JsonpAdvice** **extends** AbstractJsonpResponseBodyAdvice {

**public** JsonpAdvice() {

super("callback");

}

}

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

#### 1.4.4. Model Methods

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-modelattrib-methods)

The @ModelAttribute annotation can be used on @RequestMapping [method arguments](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-modelattrib-method-args) to create or access an Object from the model and bind it to the request. @ModelAttribute can also be used as a method-level annotation on controller methods whose purpose is not to handle requests but to add commonly needed model attributes prior to request handling.

A controller can have any number of @ModelAttribute methods. All such methods are invoked before @RequestMappingmethods in the same controller. A @ModelAttribute method can also be shared across controllers via @ControllerAdvice. See the section on [Controller Advice](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-controller-advice) for more details.

@ModelAttribute methods have flexible method signatures. They support many of the same arguments as @RequestMappingmethods except for @ModelAttribute itself nor anything related to the request body.

An example @ModelAttribute method:

@ModelAttribute

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

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

*// add more ...*

}

To add one attribute only:

@ModelAttribute

**public** Account addAccount(@RequestParam String number) {

**return** accountRepository.findAccount(number);

}

|  |  |
| --- | --- |
|  | When a name is not explicitly specified, a default name is chosen based on the Object type as explained in the Javadoc for [Conventions](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/core/Conventions.html). You can always assign an explicit name by using the overloaded addAttributemethod or through the name attribute on @ModelAttribute (for a return value). |

@ModelAttribute can also be used as a method-level annotation on @RequestMapping methods in which case the return value of the @RequestMapping method is interpreted as a model attribute. This is typically not required, as it is the default behavior in HTML controllers, unless the return value is a String which would otherwise be interpreted as a view name (also see [[mvc-coc-r2vnt]](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-coc-r2vnt)). @ModelAttribute can also help to customize the model attribute name:

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

@ModelAttribute("myAccount")

**public** Account handle() {

*// ...*

**return** account;

}

#### 1.4.5. Binder Methods

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-initbinder)

@InitBinder methods in an @Controller or @ControllerAdvice class can be used to customize type conversion for method arguments that represent String-based request values (e.g. request parameters, path variables, headers, cookies, and others). Type conversion also applies during data binding of request parameters onto @ModelAttribute arguments (i.e. command objects).

@InitBinder methods can register controller-specific java.bean.PropertyEditor, or Spring Converter and Formattercomponents. In addition, the [MVC config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-conversion) can be used to register Converter and Formatter types in a globally shared FormattingConversionService.

@InitBinder methods support many of the same arguments that a @RequestMapping methods do, except for @ModelAttribute(command object) arguments. Typically they’re are declared with a WebDataBinder argument, for registrations, and a voidreturn value. Below is an example:

@Controller

**public** **class** **FormController** {

**@InitBinder**

**public** **void** initBinder(WebDataBinder binder) {

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

dateFormat.setLenient(false);

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

}

*// ...*

}

Alternatively when using a Formatter-based setup through a shared FormattingConversionService, you could re-use the same approach and register controller-specific Formatter's:

@Controller

**public** **class** **FormController** {

**@InitBinder**

**protected** **void** initBinder(WebDataBinder binder) {

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

}

*// ...*

}

#### 1.4.6. Controller Advice

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-ann-controller-advice)

Typically @ExceptionHandler, @InitBinder, and @ModelAttribute methods apply within the @Controller class (or class hierarchy) they are declared in. If you want such methods to apply more globally, across controllers, you can declare them in a class marked with @ControllerAdvice or @RestControllerAdvice.

@ControllerAdvice is marked with @Component which means such classes can be registered as Spring beans via [component scanning](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#beans-java-instantiating-container-scan). @RestControllerAdvice is also a meta-annotation marked with both @ControllerAdvice and @ResponseBody which essentially means @ExceptionHandler methods are rendered to the response body via message conversion (vs view resolution/template rendering).

On startup, the infrastructure classes for @RequestMapping and @ExceptionHandler methods detect Spring beans of type @ControllerAdvice, and then apply their methods at runtime. Global @ExceptionHandler methods (from an @ControllerAdvice) are applied **after** local ones (from the @Controller). By contrast global @ModelAttribute and @InitBinder methods are applied **before** local ones.

By default @ControllerAdvice methods apply to every request, i.e. all controllers, but you can narrow that down to a subset of controllers via attributes on the annotation:

*// Target all Controllers annotated with @RestController*

@ControllerAdvice(annotations = RestController.class)

**public** **class** **ExampleAdvice1** {}

*// Target all Controllers within specific packages*

@ControllerAdvice("org.example.controllers")

**public** **class** **ExampleAdvice2** {}

*// Target all Controllers assignable to specific classes*

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

**public** **class** **ExampleAdvice3** {}

Keep in mind the above selectors are evaluated at runtime and may negatively impact performance if used extensively. See the[@ControllerAdvice](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html) Javadoc for more details.

### 1.5. URI Links

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 subclass 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()

|  |  |
| --- | --- |
|  | 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. Note that you can also use the[ForwardedHeaderFilter](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#filters-forwarded-headers) to the same once, globally. |

#### 1.5.1. Links to Controllers

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();

#### 1.5.2. Links in 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 yourRequestMappingHandlerMapping. 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.

### 1.6. Exception Handling

#### 1.6.1. Overview

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.

#### 1.6.2. @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 subclasses). 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.

|  |  |
| --- | --- |
|  | For @ExceptionHandler methods, a root exception match will be preferred to just matching a cause of the current exception, among the handler methods of a particular controller or advice bean. However, a cause match on a higher-priority @ControllerAdvice will still be preferred to a any match (whether root or cause level) on a lower-priority advice bean. As a consequence, when using a multi-advice arrangement, please declare your primary root exception mappings on a prioritized advice bean with a corresponding order! |

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.

#### 1.6.3. Framework exceptions

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

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

| **Exception** | **HTTP Status Code** |
| --- | --- |
| BindException | 400 (Bad Request) |
| ConversionNotSupportedException | 500 (Internal Server Error) |
| HttpMediaTypeNotAcceptableException | 406 (Not Acceptable) |
| HttpMediaTypeNotSupportedException | 415 (Unsupported Media Type) |
| HttpMessageNotReadableException | 400 (Bad Request) |
| HttpMessageNotWritableException | 500 (Internal Server Error) |
| HttpRequestMethodNotSupportedException | 405 (Method Not Allowed) |
| MethodArgumentNotValidException | 400 (Bad Request) |
| 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.

#### 1.6.4. REST API exceptions

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.

#### 1.6.5. Annotated Exception

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.

#### 1.6.6. 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") %>

}

### 1.7. Async Requests

Spring MVC has an extensive integration with the Servlet 3.0 asynchronous request processing. [DeferredResult](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-deferredresult) and [Callable](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-callable)provide basic support for producing return values asynchronously. Controllers can produce [response streams](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-http-streaming) including [SSE](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-sse) and [raw data](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-output-stream). Controllers can use reactive clients and return [reactive return types](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-reactive-types) to Spring MVC for response handling.

#### 1.7.1. DeferredResult

Once the asynchronous request processing feature is [enabled](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-configuration) in the Servlet container, controller methods can wrap any supported controller method return value with DeferredResult:

@GetMapping("/quotes")

@ResponseBody

**public** DeferredResult<String> quotes() {

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

*// Save the deferredResult somewhere..*

**return** deferredResult;

}

*// From some other thread...*

deferredResult.setResult(data);

The controller can produce the return value asynchronously, from a different thread, for example in response to an external event (JMS message), a scheduled task, or other.

#### 1.7.2. Callable

A controller may also wrap any supported return value with java.util.concurrent.Callable:

@PostMapping

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

**return** **new** Callable<String>() {

**public** String call() **throws** Exception {

*// ...*

**return** "someView";

}

};

}

The return value will then be obtained by executing the the given task through the [configured](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-configuration-spring-mvc) TaskExecutor.

#### 1.7.3. Processing

Here is a very concise overview of Servlet asynchronous request processing:

* 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.

DeferredResult processing:

* Controller returns a DeferredResult and saves it in some in-memory queue or list where it can be accessed.
* Spring MVC calls request.startAsync().
* Meanwhile 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 return value.

Callable processing:

* Controller returns a Callable.
* Spring MVC calls request.startAsync() and submits the Callable to a TaskExecutor for processing in a separate thread.
* Meanwhile the DispatcherServlet and all Filter’s exit the Servlet container thread but the response remains open.
* Eventually the Callable produces a result and Spring MVC dispatches the request back to the Servlet container to complete processing.
* The DispatcherServlet is invoked again and processing resumes with the asynchronously produced return value from the Callable.

For further background and context you can also read [the blog posts](https://spring.io/blog/2012/05/07/spring-mvc-3-2-preview-introducing-servlet-3-async-support) that introduced asynchronous request processing support in Spring MVC 3.2.

#### 1.7.4. Exception handling

When using a DeferredResult you can choose whether to call setResult or setErrorResult with an exception. In both cases Spring MVC dispatches the request back to the Servlet container to complete processing. It is then treated either as if the controller method returned the given value, or as if it produced the given exception. The exception then goes through the regular exception handling mechanism, e.g. invoking @ExceptionHandler methods.

When using Callable, similar processing logic follows. The main difference being that the result is returned from the Callableor an exception is raised by it.

#### 1.7.5. Interception

HandlerInterceptor's can also be AsyncHandlerInterceptor in order to receive the afterConcurrentHandlingStarted callback on the initial request that starts asynchronous processing instead of postHandle and afterCompletion.

HandlerInterceptor's can also register a CallableProcessingInterceptor or a DeferredResultProcessingInterceptor in order to integrate more deeply with the lifecycle of an asynchronous request for example to handle a timeout event. See[AsyncHandlerInterceptor](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/AsyncHandlerInterceptor.html) for more details.

DeferredResult provides onTimeout(Runnable) and onCompletion(Runnable) callbacks. See the Javadoc of DeferredResult for more details. Callable can be substituted for WebAsyncTask that exposes additional methods for timeout and completion callbacks.

#### 1.7.6. Streaming response

What if you wanted to push multiple events on a single HTTP response? The ResponseBodyEmitter return value can be used to stream multiple Objects, where each Object sent is serialized with an [HttpMessageConverter](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/integration.html#rest-message-conversion) and written to the response. For example:

@GetMapping("/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();

ResponseBodyEmitter can also be used as the body in a ResponseEntity allowing you to customize the status and headers of the response.

#### 1.7.7. Server-Sent Events

SseEmitter is a sub-class of ResponseBodyEmitter that provides support for [Server-Sent Events](https://www.w3.org/TR/eventsource/) where events sent from the server are formatted according to the W3C SSE specification. In order to produce an SSE stream from a controller simply return SseEmitter:

@GetMapping(path="/events", produces=MediaType.TEXT\_EVENT\_STREAM\_VALUE)

**public** SseEmitter handle() {

SseEmitter emitter = **new** SseEmitter();

*// 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();

While SSE is the main option for streaming into browsers, note that Internet Explorer does not support Server-Sent Events. Consider using Spring’s [WebSocket messaging](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket) with [SockJS fallback](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-fallback) transports (including SSE) that target a wide range of browsers.

#### 1.7.8. Streaming raw data

Sometimes it is useful to bypass message conversion and stream directly to the response OutputStream for example for a file download. Use the of the StreamingResponseBody return value type to do that:

@GetMapping("/download")

**public** StreamingResponseBody handle() {

**return** **new** StreamingResponseBody() {

@Override

**public** **void** writeTo(OutputStream outputStream) **throws** IOException {

*// write...*

}

};

}

StreamingResponseBody can be used as the body in a ResponseEntity allowing you to customize the status and headers of the response.

#### 1.7.9. Reactive return values

Spring MVC supports use of reactive client libraries in a controller. This includes the WebClient from spring-webflux and others such as Spring Data reactive data repositories. In such scenarios it is convenient to be able to return reactive types from the controller method .

Reactive return values are handled as follows:

* A single-value promise is adapted to and similar to using DeferredResult. Examples include Mono (Reactor) or Single(RxJava).
* A multi-value stream, with a streaming media type such as "application/stream+json" or "text/event-stream", is adapted to and similar to using ResponseBodyEmitter or SseEmitter. Examples include Flux (Reactor) or Observable (RxJava). Applications can also return Flux<ServerSentEvent> or Observable<ServerSentEvent>.
* A multi-value stream, with any other media type (e.g. "application/json"), is adapted to and similar to using DeferredResult<List<?>>.

|  |  |
| --- | --- |
|  | Spring MVC supports Reactor and RxJava through the [ReactiveAdapterRegistry](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/core/ReactiveAdapterRegistry.html) from spring-core which allows it to adapt from multiple reactive libraries. |

When streaming to the response with a reactive type, Spring MVC performs (blocking) writes to the response through the through the [configured](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-async-configuration-spring-mvc) MVC TaskExecutor. By default this is a SyncTaskExecutor and not suitable for production. [SPR-16203](https://jira.spring.io/browse/SPR-16203)will provide better defaults. In the mean time please configure the executor through the MVC config.

#### 1.7.10. Configuration

For asynchronous requests there are minor requirements at the Servlet container level and more controls in Spring MVC configuration.

##### Servlet container config

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 config

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 Callableinstances 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.

### 1.8. CORS

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-cors)

#### 1.8.1. Introduction

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-cors-intro)

For security reasons browsers prohibit AJAX calls to resources outside the current origin. For example you could have your bank account in one tab and evil.com in another. Scripts from evil.com should not be able to make AJAX requests to your bank API with your credentials, e.g. withdrawing money from your account!

Cross-Origin Resource Sharing (CORS) is a [W3C specification](https://www.w3.org/TR/cors/) implemented by [most browsers](https://caniuse.com/#feat=cors) that allows you to specify what kind of cross domain requests are authorized rather than using less secure and less powerful workarounds based on IFRAME or JSONP.

#### 1.8.2. Processing

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-cors-processing)

The CORS specification distinguishes between preflight, simple, and actual requests. To learn how CORS works, you can read[this article](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS), among many others, or refer to the specification for more details.

Spring MVC HandlerMapping's provide built-in support for CORS. After successfully mapping a request to a handler, HandlerMapping's check the CORS configuration for the given request and handler and take further actions. Preflight requests are handled directly while simple and actual CORS requests are intercepted, validated, and have required CORS response headers set.

In order to enable cross-origin requests (i.e. the Origin header is present and differs from the host of the request) you need to have some explicitly declared CORS configuration. If no matching CORS configuration is found, preflight requests are rejected. No CORS headers are added to the responses of simple and actual CORS requests and consequently browsers reject them.

Each HandlerMapping can be [configured](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/handler/AbstractHandlerMapping.html#setCorsConfigurations-java.util.Map-) individually with URL pattern based CorsConfiguration mappings. In most cases applications will use the MVC Java config or the XML namespace to declare such mappings, which results in a single, global map passed to all HadlerMappping's.

Global CORS configuration at the HandlerMapping level can be combined with more fine-grained, handler-level CORS configuration. For example annotated controllers can use class or method-level @CrossOrigin annotations (other handlers can implement CorsConfigurationSource).

The rules for combining global and local configuration are generally additive — e.g. all global and all local origins. For those attributes where only a single value can be accepted such as allowCredentials and maxAge, the local overrides the global value. See [CorsConfiguration#combine(CorsConfiguration)](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/cors/CorsConfiguration.html#combine-org.springframework.web.cors.CorsConfiguration-) for more details.

|  |  |
| --- | --- |
|  | To learn more from the source or make advanced customizations, check:   * CorsConfiguration * CorsProcessor, DefaultCorsProcessor * AbstractHandlerMapping |

#### 1.8.3. @CrossOrigin

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-cors-controller)

The [@CrossOrigin](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/bind/annotation/CrossOrigin.html) annotation enables cross-origin requests on annotated controller methods:

@RestController

@RequestMapping("/account")

**public** **class** **AccountController** {

@CrossOrigin

@GetMapping("/{id}")

**public** Account retrieve(@PathVariable Long id) {

*// ...*

}

@DeleteMapping("/{id}")

**public** **void** remove(@PathVariable Long id) {

*// ...*

}

}

By default @CrossOrigin allows:

* All origins.
* All headers.
* All HTTP methods to which the controller method is mapped.
* allowedCredentials is not enabled by default since that establishes a trust level that exposes sensitive user-specific information such as cookies and CSRF tokens, and should only be used where appropriate.
* maxAge is set to 30 minutes.

@CrossOrigin is supported at the class level too and inherited by all methods:

@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)

@RestController

@RequestMapping("/account")

**public** **class** **AccountController** {

@GetMapping("/{id}")

**public** Account retrieve(@PathVariable Long id) {

*// ...*

}

@DeleteMapping("/{id}")

**public** **void** remove(@PathVariable Long id) {

*// ...*

}

}

CrossOrigin can be used at both class and method-level:

@CrossOrigin(maxAge = 3600)

@RestController

@RequestMapping("/account")

**public** **class** **AccountController** {

@CrossOrigin("http://domain2.com")

@GetMapping("/{id}")

**public** Account retrieve(@PathVariable Long id) {

*// ...*

}

@DeleteMapping("/{id}")

**public** **void** remove(@PathVariable Long id) {

*// ...*

}

}

#### 1.8.4. Global Config

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-cors-global)

In addition to fine-grained, controller method level configuration you’ll probably want to define some global CORS configuration too. You can set URL-based CorsConfiguration mappings individually on any HandlerMapping. Most applications however will use the MVC Java config or the MVC XNM namespace to do that.

By default global configuration enables the following:

* All origins.
* All headers.
* GET, HEAD, and POST methods.
* allowedCredentials is not enabled by default since that establishes a trust level that exposes sensitive user-specific information such as cookies and CSRF tokens, and should only be used where appropriate.
* maxAge is set to 30 minutes.

##### Java Config

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-cors-global)

To enable CORS in the MVC Java config, use the CorsRegistry callback:

@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(true).maxAge(3600);

*// Add more mappings...*

}

}

##### XML Config

To enable CORS in the XML namespace, use the <mvc:cors> element:

<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="true"

max-age="123" />

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

allowed-origins="http://domain1.com" />

</mvc:cors>

#### 1.8.5. CORS Filter

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-cors-webfilter)

You can apply CORS support through the built-in [CorsFilter](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/filter/CorsFilter.html).

|  |  |
| --- | --- |
|  | If you’re trying to use the CorsFilter with Spring Security, keep in mind that Spring Security has [built-in support](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#cors) for CORS. |

To configure the filter pass a CorsConfigurationSource to its constructor:

CorsConfiguration config = **new** CorsConfiguration();

*// Possibly...*

*// config.applyPermitDefaultValues()*

config.setAllowCredentials(true);

config.addAllowedOrigin("http://domain1.com");

config.addAllowedHeader("**");**

**config.addAllowedMethod("**");

UrlBasedCorsConfigurationSource source = **new** UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", config);

CorsFilter filter = **new** CorsFilter(source);

### 1.9. Web Security

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-web-security)

The [Spring Security](https://projects.spring.io/spring-security/) project provides support for protecting web applications from malicious exploits. Check out the Spring Security reference documentation including:

* [Spring MVC Security](https://docs.spring.io/spring-security/site/docs/current/reference/html5/#mvc)
* [Spring MVC Test Support](https://docs.spring.io/spring-security/site/docs/current/reference/html5/#test-mockmvc)
* [CSRF protection](https://docs.spring.io/spring-security/site/docs/current/reference/html5/#csrf)
* [Security Response Headers](https://docs.spring.io/spring-security/site/docs/current/reference/html5/#headers)

[HDIV](http://hdiv.org/) is another web security framework that integrates with Spring MVC.

### 1.10. HTTP Caching

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](https://en.wikipedia.org/wiki/HTTP_ETag) (entity tag) is an HTTP response header returned by an HTTP/1.1 compliant web server used to determine change in content at a given URL. It can be considered to be the more sophisticated successor to the Last-Modified header. When a server returns a representation with an ETag header, the client can use this header in subsequent GETs, in an If-None-Matchheader. 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.

#### 1.10.1. Cache-Control

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

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

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

The [CacheControl](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/http/CacheControl.html) builder class simply describes the available "Cache-Control" directives and makes it easier to build your own HTTP caching strategy. Once built, a CacheControl instance can then be accepted as an argument in several Spring Web MVC APIs.

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

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

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

CacheControl ccNoStore = CacheControl.noStore();

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

*// public caches should not transform the response*

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

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

.noTransform().cachePublic();

#### 1.10.2. Static resources

Static resources should be served with appropriate 'Cache-Control' and conditional headers for optimal performance.[Configuring a ResourceHttpRequestHandler](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-static-resources) for serving static resources not only natively writes 'Last-Modified' headers by reading a file’s metadata, but also 'Cache-Control' headers if properly configured.

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

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **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>

#### 1.10.3. @Controller caching

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 [HttpEntity](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-httpentity), controllers can interact with the request/response using HttpEntity types. Controllers returning ResponseEntity can include HTTP caching information in responses like this:

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

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

Book book = findBook(id);

String version = book.getVersion();

**return** ResponseEntity

.ok()

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

.eTag(version) *// lastModified is also available*

.body(book);

}

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

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

@RequestMapping

**public** String myHandleMethod(WebRequest webRequest, Model model) {

**long** lastModified = *// 1. application-specific calculation*

**if** (request.checkNotModified(lastModified)) {

*// 2. shortcut exit - no further processing necessary*

**return** null;

}

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

model.addAttribute(...);

**return** "myViewName";

}

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

Note that there are 3 variants for this:

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

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

#### 1.10.4. ETag Filter

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 by caching the content written to the response and generating an MD5 hash over that to send as an ETag header. 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, lets the request be processed as usual, and at the end 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 avoid computation.

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

### 1.11. MVC Config

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config)

The MVC Java config and the MVC XML namespace provide default configuration suitable for most applications along with a configuration API to customize it.

For more advanced customizations, not available in the configuration API, see [Advanced Java Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-advanced-java) and [Advanced XML Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-config-advanced-xml).

You do not need to understand the underlying beans created by the MVC Java config and the MVC namespace but if you want to learn more, see [Special Bean Types](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-servlet-special-bean-types) and [Framework Config](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-servlet-config).

#### 1.11.1. Enable MVC Config

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-enable)

In Java config use the @EnableWebMvc annotation:

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** {

}

In XML use the <mvc:annotation-driven> element:

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

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

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

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

xsi:schemaLocation="

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

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

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

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

<mvc:annotation-driven/>

</beans>

The above registers a number of Spring MVC [infrastructure beans](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-servlet-special-bean-types) also adapting to dependencies available on the classpath: e.g. payload converters for JSON, XML, etc.

#### 1.11.2. MVC Config API

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-customize)

In Java config implement WebMvcConfigurer interface:

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

*// Implement configuration methods...*

}

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

#### 1.11.3. Type conversion

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-conversion)

By default formatters for Number and Date types are installed, including support for the @NumberFormat and @DateTimeFormatannotations. Full support for the Joda Time formatting library is also installed if Joda Time is present on the classpath.

In Java config, register custom formatters and converters:

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addFormatters(FormatterRegistry registry) {

*// ...*

}

}

In XML, the same:

<?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](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#format-FormatterRegistrar-SPI) and the FormattingConversionServiceFactoryBean for more information on when to use FormatterRegistrars. |

#### 1.11.4. Validation

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-validation)

By default if [Bean Validation](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#validation-beanvalidation-overview) is present on the classpath — e.g. Hibernate Validator, the LocalValidatorFactoryBean is registered as a global [Validator](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/core.html#validator) for use with @Valid and Validated on controller method arguments.

In Java config, you can customize the global Validator instance:

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** Validator getValidator(); {

*// ...*

}

}

In XML, the same:

<?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>

Note that you can also register Validator's locally:

@Controller

**public** **class** **MyController** {

@InitBinder

**protected** **void** initBinder(WebDataBinder binder) {

binder.addValidators(**new** FooValidator());

}

}

|  |  |
| --- | --- |
|  | If you need to have a LocalValidatorFactoryBean injected somewhere, create a bean and mark it with @Primaryin order to avoid conflict with the one declared in the MVC config. |

#### 1.11.5. Interceptors

In Java config, register interceptors to apply to incoming requests:

@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/\*");

}

}

In XML, the same:

<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>

#### 1.11.6. Content Types

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-content-negotiation)

You can configure how Spring MVC determines the requested media types from the request — e.g. Accept header, URL path extension, query parameter, etc.

By default the URL path extension is checked first — with json, xml, rss, and atom registered as known extensions depending on classpath dependencies, and the "Accept" header is checked second.

Consider changing those defaults to Accept header only and if you must use URL-based content type resolution consider the query parameter strategy over the path extensions. See [Suffix match](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-suffix-pattern-match) and [Suffix match and RFD](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-rfd) for more details.

In Java config, customize requested content type resolution:

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureContentNegotiation(ContentNegotiationConfigurer configurer) {

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

}

}

In XML, the same:

<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>

#### 1.11.7. Message Converters

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-message-codecs)

Customization of HttpMessageConverter can be achieved in Java config by overriding [configureMessageConverters()](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurer.html#configureMessageConverters-java.util.List-) if you want to replace the default converters created by Spring MVC, or by overriding [extendMessageConverters()](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurer.html#extendMessageConverters-java.util.List-) if you just want to customize them or add additional converters to the default ones.

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

@Configuration

@EnableWebMvc

**public** **class** **WebConfiguration** **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](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/http/converter/json/Jackson2ObjectMapperBuilder.html) is used to create a common configuration for both MappingJackson2HttpMessageConverter and MappingJackson2XmlHttpMessageConverter with indentation enabled, a customized date format and the registration of [jackson-module-parameter-names](https://github.com/FasterXML/jackson-module-parameter-names) that adds support for accessing parameter names (feature added in Java 8).

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

1. [DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES](https://fasterxml.github.io/jackson-databind/javadoc/2.6/com/fasterxml/jackson/databind/DeserializationFeature.html#FAIL_ON_UNKNOWN_PROPERTIES) is disabled.
2. [MapperFeature.DEFAULT\_VIEW\_INCLUSION](https://fasterxml.github.io/jackson-databind/javadoc/2.6/com/fasterxml/jackson/databind/MapperFeature.html#DEFAULT_VIEW_INCLUSION) is disabled.

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

1. [jackson-datatype-jdk7](https://github.com/FasterXML/jackson-datatype-jdk7): support for Java 7 types like java.nio.file.Path.
2. [jackson-datatype-joda](https://github.com/FasterXML/jackson-datatype-joda): support for Joda-Time types.
3. [jackson-datatype-jsr310](https://github.com/FasterXML/jackson-datatype-jsr310): support for Java 8 Date & Time API types.
4. [jackson-datatype-jdk8](https://github.com/FasterXML/jackson-datatype-jdk8): support for other Java 8 types like Optional.

|  |  |
| --- | --- |
|  | Enabling indentation with Jackson XML support requires [woodstox-core-asl](https://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.codehaus.woodstox%22%20AND%20a%3A%22woodstox-core-asl%22) dependency in addition to [jackson-dataformat-xml](https://search.maven.org/#search%7Cga%7C1%7Ca%3A%22jackson-dataformat-xml%22) one. |

Other interesting Jackson modules are available:

1. [jackson-datatype-money](https://github.com/zalando/jackson-datatype-money): support for javax.money types (unofficial module)
2. [jackson-datatype-hibernate](https://github.com/FasterXML/jackson-datatype-hibernate): support for Hibernate specific types and properties (including lazy-loading aspects)

It is also possible to do the same in XML:

<mvc:annotation-driven>

<mvc:message-converters>

<bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">

<property name="objectMapper" ref="objectMapper"/>

</bean>

<bean class="org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter">

<property name="objectMapper" ref="xmlMapper"/>

</bean>

</mvc:message-converters>

</mvc:annotation-driven>

<bean id="objectMapper" class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean"

p:indentOutput="true"

p:simpleDateFormat="yyyy-MM-dd"

p:modulesToInstall="com.fasterxml.jackson.module.paramnames.ParameterNamesModule"/>

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

#### 1.11.8. 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"/>

#### 1.11.9. View Resolvers

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-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;

}

}

#### 1.11.10. Static Resources

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-static-resources)

This option provides a convenient way to serve static resources from a list of [Resource](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/core/io/Resource.html)-based locations.

In the example below, given a request that starts with "/resources", the relative path is used to find and serve static resources relative to "/public" under the web application root or on the classpath under "/static". The resources are served with a 1-year future expiration to ensure maximum use of the browser cache and a reduction in HTTP requests made by the browser. The Last-Modified header is also evaluated and if present a 304 status code is returned.

In Java config:

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

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

.addResourceLocations("/public", "classpath:/static/")

.setCachePeriod(31556926);

}

}

In XML:

<mvc:resources mapping="/resources/\*\*"

location="/public, classpath:/static/"

cache-period="31556926" />

See also [HTTP caching support for static resources](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-caching-static-resources).

The resource handler also supports a chain of [ResourceResolver](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/resource/ResourceResolver.html)'s and [ResourceResolver](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/resource/ResourceTransformer.html)'s. which can be used to create a toolchain for working with optimized resources.

The VersionResourceResolver can be used for versioned resource URLs based on an MD5 hash computed from the content, a fixed application version, or other. A ContentVersionStrategy (MD5 hash) is a good choice with some notable exceptions such as JavaScript resources used with a module loader.

For example in Java config;

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

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

.addResourceLocations("/public/")

.resourceChain(true)

.addResolver(**new** VersionResourceResolver().addContentVersionStrategy("/\*\*"));

}

}

In XML, the same:

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

<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>

You can use ResourceUrlProvider to rewrite URLs and apply the full chain of resolvers and transformers — e.g. to insert versions. The MVC config provides a ResourceUrlProvider bean so it can be injected into others. You can also make the rewrite transparent with the ResourceUrlEncodingFilter for Thymeleaf, JSPs, FreeMarker, and others with URL tags that rely on HttpServletResponse#encodeURL.

[WebJars](http://www.webjars.org/documentation) is also supported via WebJarsResourceResolver and automatically registered when "org.webjars:webjars-locator" is present on the classpath. The resolver can re-write URLs to include the version of the jar and can also match to incoming URLs without versions — e.g. "/jquery/jquery.min.js" to "/jquery/1.2.0/jquery.min.js".

#### 1.11.11. Default Servlet

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 aDefaultServletHttpRequestHandler 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"/>

#### 1.11.12. Path Matching

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-path-matching)

This allows customizing options related to URL matching and treatment of the URL. For details on the individual options check out the [PathMatchConfigurer](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/config/annotation/PathMatchConfigurer.html) API.

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() {

*//...*

}

}

In XML, the same:

<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"/>

#### 1.11.13. Advanced Java Config

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-config-advanced-java)

@EnableWebMvc imports DelegatingWebMvcConfiguration that (1) provides default Spring configuration for Spring MVC applications and (2) detects and delegates to WebMvcConfigurer's to customize that configuration.

For advanced mode, remove @EnableWebMvc and extend directly from DelegatingWebMvcConfiguration instead of implementing WebMvcConfigurer:

@Configuration

**public** **class** **WebConfig** **extends** DelegatingWebMvcConfiguration {

*// ...*

}

You can keep existing methods in WebConfig but you can now also override bean declarations from the base class and you can still have any number of other WebMvcConfigurer's on the classpath.

#### 1.11.14. Advanced XML Config

The MVC namespace does not have an advanced mode. If you need to customize a property on a bean that you can’t change otherwise, you can use the BeanPostProcessor lifecycle hook of the Spring ApplicationContext:

@Component

**public** **class** **MyPostProcessor** **implements** BeanPostProcessor {

**public** Object postProcessBeforeInitialization(Object bean, String name) **throws** BeansException {

*// ...*

}

}

Note that MyPostProcessor needs to be declared as a bean either explicitly in XML or detected through a <component scan/>declaration.

### 1.12. View Technologies

#### 1.12.1. 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 [View Resolution](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-viewresolver) which covers the basics of how views in general are coupled to the MVC framework.

#### 1.12.2. Thymeleaf

[Thymeleaf](http://www.thymeleaf.org/) 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](http://www.thymeleaf.org/documentation.html) documentation section for more details.

#### 1.12.3. Groovy Markup

The [Groovy Markup Template Engine](http://groovy-lang.org/templating.html#_the_markuptemplateengine) 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.

##### 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/"/>

##### 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')

}

}

#### 1.12.4. FreeMarker

[FreeMarker](http://www.freemarker.org/) 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](http://www.freemarker.org/) web site.

##### 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!

##### 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. |

##### 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.

##### Advanced config

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 ajava.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.

##### 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.

###### Input 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 5. 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/> |
| **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 "<br>").
* 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 <b></b> 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

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 -->

#### 1.12.5. 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](http://www.mularien.com/blog/2008/04/24/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. |

##### 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 theResourceBundleViewResolver. 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.

##### 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.

##### Spring’s JSP tag library

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 haveHTML escaping features to enable or disable escaping of characters.

The spring.tld tag library descriptor (TLD) is included in the spring-webmvc.jar. For a comprehensive reference on individual tags, browse the [API reference](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/tags/package-summary.html#package.description) or see the tag library description.

##### 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](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-view-jsp-formtaglib-formtag). 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 notchecked, 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 optionand 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>

The spring-form.tld tag library descriptor (TLD) is included in the spring-webmvc.jar. For a comprehensive reference on individual tags, browse the [API reference](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/servlet/tags/form/package-summary.html#package.description) or see the tag library description.

###### 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.

#### 1.12.6. Script views

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 Invocableinterfaces.

It has been tested with:

* [Handlebars](http://handlebarsjs.com/) running on [Nashorn](http://openjdk.java.net/projects/nashorn/)
* [Mustache](https://mustache.github.io/) running on [Nashorn](http://openjdk.java.net/projects/nashorn/)
* [React](https://facebook.github.io/react/) running on [Nashorn](http://openjdk.java.net/projects/nashorn/)
* [EJS](http://www.embeddedjs.com/) running on [Nashorn](http://openjdk.java.net/projects/nashorn/)
* [ERB](http://www.stuartellis.eu/articles/erb/) running on [JRuby](http://jruby.org/)
* [String templates](https://docs.python.org/2/library/string.html#template-strings) running on [Jython](http://www.jython.org/)

##### Requirements

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

* [Nashorn](http://openjdk.java.net/projects/nashorn/) Javascript engine is provided builtin with Java 8+. Using the latest update release available is highly recommended.
* [Rhino](http://docs.oracle.com/javase/7/docs/technotes/guides/scripting/programmer_guide/#jsengine) 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](http://jruby.org/) dependency should be added in order to get Ruby support.
* [Jython](http://www.jython.org/) 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](http://www.webjars.org/) to add Maven/Gradle dependencies in order to make your javascript libraries available in the classpath.

##### Script templates

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](http://handlebarsjs.com/) needs to compile templates before using them, and requires a [polyfill](https://en.wikipedia.org/wiki/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](https://bugs.openjdk.java.net/browse/JDK-8076099). |

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](https://github.com/spring-projects/spring-framework/tree/master/spring-webmvc/src/test/java/org/springframework/web/servlet/view/script), [resources](https://github.com/spring-projects/spring-framework/tree/master/spring-webmvc/src/test/resources/org/springframework/web/servlet/view/script)) for more configuration examples.

#### 1.12.7. XML Marshalling

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](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#oxm).

#### 1.12.8. 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.tiles3package. |

##### 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](https://tiles.apache.org/framework/dependency-management.html) to your project.

##### Configuration

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](https://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 view names and view classes 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>

#### 1.12.9. 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.

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 [Annotated Controllers](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-controller) for details of Spring Web MVC’sController interface. The XSLT Controller will turn the list of words into a simple XML document ready for transformation.

##### Beans

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.

##### Controller

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.

##### 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>

#### 1.12.10. Document views: PDF, Excel

##### 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 Apache POI library to your classpath, and for PDF generation, the common iText 2.1.7 or its fork OpenPDF (e.g. OpenPDF 1.0.4).

##### Configuration

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).

##### View definition

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

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!

##### 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 oforg.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.

##### PDF views

The PDF version of the word list is even simpler. This time, the class extendsorg.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.

#### 1.12.11. Feed views: RSS, Atom

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](https://rome.dev.java.net/) 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](https://spring.io/blog/2009/03/16/adding-an-atom-view-to-an-application-using-spring-s-rest-support).

#### 1.12.12. Jackson

##### JSON

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](https://en.wikipedia.org/wiki/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.

##### XML

The MappingJackson2XmlView uses the [Jackson XML extension](https://github.com/FasterXML/jackson-dataformat-xml)'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.

### 1.13. HTTP/2

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-http2)

Servlet 4 containers are required to support HTTP/2 and Spring Framework 5 is compatible with Servlet API 4. From a programming model perspective there is nothing specific that applications need to do. However there are considerations related to server configuration. For more details please check out the [HTTP/2 wiki page](https://github.com/spring-projects/spring-framework/wiki/HTTP-2-support).

The Servlet API does expose one construct related to HTTP/2. The javax.servlet.http.PushBuilder can used to proactively push resources to clients and it is supported as a [method argument](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-ann-arguments) to @RequestMapping methods.

## 2. REST Clients

This section describes options for client-side access to REST endpoints.

### 2.1. RestTemplate

RestTemplate is the original Spring REST client that follows a similar approach to other template classes in the Spring Framework (e.g. JdbcTemplate, JmsTemplate, etc.) by providing a list of parameterizable methods to perform HTTP requests.

RestTemplate has a synchronous API and relies on blocking I/O. This is okay for client scenarios with low concurrency. In a server environment or when orchestrating a sequence of remote calls, prefer using the WebClient which provides a more efficient execution model including seamless support for streaming.

See [RestTemplate](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/integration.html#rest-client-access) for more details on using the RestTemplate.

### 2.2. WebClient

WebClient is a reactive client that provides an alternative to the RestTemplate. It exposes a functional, fluent API and relies on non-blocking I/O which allows it to support high concurrency more efficiently (i.e. using a small number of threads) than theRestTemplate. WebClient is a natural fit for streaming scenarios.

See [WebClient](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-client) for more details on using the WebClient.

## 3. Testing

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-test)

### 3.1. Servlet API Mocks

spring-test provides mock implementations of Servlet API contracts for unit testing controllers, filters, and other web components. See [Servlet API](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/testing.html#mock-objects-servlet) mock objects for more details.

### 3.2. TestContext Framework

spring-test provides support for loading Spring configuration in JUnit and TestNG tests including efficient caching of the loaded configuration across test methods and support for loading a WebApplicationContext with a MockServletContext. See [TestContext Framework](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/testing.html#testcontext-framework) for more details.

### 3.3. Spring MVC Test (MockMvc)

spring-test provides a framework for testing annotated controllers through the DispatcherServlet, i.e. supporting annotations and complete with Spring MVC infrastructure, but without an HTTP server. See [Spring MVC Test](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/testing.html#spring-mvc-test-framework) for more details.

### 3.4. Client-side REST

spring-test provides a MockRestServiceServer that can be used as a mock server for testing client-side code that internally uses the RestTemplate. See [Client REST Tests](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/testing.html#spring-mvc-test-client) for more details.

### 3.5. WebTestClient

spring-test provides a dedicated WebTestClient for end-to-end integration testing. WebTestClient can connect to any server over an HTTP connection. It is non-blocking, reactive client and well suited for testing asynchronous and streaming scenarios.

## 4. WebSockets

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-websocket)

This part of the reference documentation covers support for Servlet stack, WebSocket messaging that includes raw WebSocket interactions, WebSocket emulation via SockJS, and pub-sub messaging via STOMP as a sub-protocol over WebSocket.

### 4.1. Introduction

The WebSocket protocol [RFC 6455](https://tools.ietf.org/html/rfc6455) provides a standardized way to establish a full-duplex, two-way communication channel between client and server over a single TCP connection. It is a different TCP protocol from HTTP but is designed to work over HTTP, using ports 80 and 443 and allowing re-use of existing firewall rules.

A WebSocket interaction begins with an HTTP request that uses the HTTP "Upgrade" header to upgrade, or in this case to switch, to the WebSocket protocol:

GET /spring-websocket-portfolio/portfolio HTTP/1.1

Host: localhost:8080

**Upgrade: websocket**

**Connection: Upgrade**

Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==

Sec-WebSocket-Protocol: v10.stomp, v11.stomp

Sec-WebSocket-Version: 13

Origin: http://localhost:8080

Instead of the usual 200 status code, a server with WebSocket support returns:

**HTTP/1.1 101 Switching Protocols**

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=

Sec-WebSocket-Protocol: v10.stomp

After a successful handshake the TCP socket underlying the HTTP upgrade request remains open for both client and server to continue to send and receive messages.

A complete introduction of how WebSockets work is beyond the scope of this document. Please read RFC 6455, the WebSocket chapter of HTML5, or one of many introductions and tutorials on the Web.

Note that if a WebSocket server is running behind a web server (e.g. nginx) you will likely need to configure it to pass WebSocket upgrade requests on to the WebSocket server. Likewise if the application runs in a cloud environment, check the instructions of the cloud provider related to WebSocket support.

#### 4.1.1. HTTP vs WebSocket

Even though WebSocket is designed to be HTTP compatible and starts with an HTTP request, it is important to understand that the two protocols lead to very different architectures and application programming models.

In HTTP and REST, an application is modeled as many URLs. To interact with the application clients access those URLs, request-response style. Servers route requests to the appropriate handler based on the HTTP URL, method, and headers.

By contrast in WebSockets there is usually just one URL for the initial connect and subsequently all application messages flow on that same TCP connection. This points to an entirely different asynchronous, event-driven, messaging architecture.

WebSocket is also a low-level transport protocol which unlike HTTP does not prescribe any semantics to the content of messages. That means there is no way to route or process a message unless client and server agree on message semantics.

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol (e.g. STOMP), via the "Sec-WebSocket-Protocol" header on the HTTP handshake request, or in the absence of that they need to come up with their own conventions.

#### 4.1.2. When to use it?

WebSockets can make a web page dynamic and interactive. However in many cases a combination of Ajax and HTTP streaming and/or long polling could provide a simple and effective solution.

For example news, mail, and social feeds need to update dynamically but it may be perfectly okay to do so every few minutes. Collaboration, games, and financial apps on the other hand need to be much closer to real time.

Latency alone is not a deciding factor. If the volume of messages is relatively low (e.g. monitoring network failures) HTTP streaming or polling may provide an effective solution. It is the combination of low latency, high frequency and high volume that make the best case for the use WebSocket.

Keep in mind also that over the Internet, restrictive proxies outside your control, may preclude WebSocket interactions either because they are not configured to pass on the Upgrade header or because they close long lived connections that appear idle? This means that the use of WebSocket for internal applications within the firewall is a more straight-forward decision than it is for public facing applications.

### 4.2. WebSocket API

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-websocket-server)

The Spring Framework provides a WebSocket API that can be used to write client and server side applications that handle WebSocket messages.

#### 4.2.1. WebSocketHandler

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-websocket-server-handler)

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](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-servlet). 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](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/socket/server/support/WebSocketHttpRequestHandler.html).

#### 4.2.2. WebSocket Handshake

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-websocket-server-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](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-server-deployment) for more on this subject). Both the Java-config and XML namespace make it possible to configure a custom HandshakeHandler.

|  |  |
| --- | --- |
|  | 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. |

#### 4.2.3. Deployment

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 server-specific RequestUpgradeStrategy's even when running in a JSR-356 runtime. Such strategies currently exist for Tomcat, Jetty, GlassFish, WebLogic, WebSphere, and Undertow (and WildFly).

|  |  |
| --- | --- |
|  | A request to overcome the above limitation in the Java WebSocket API has been created and can be followed at[WEBSOCKET\_SPEC-211](https://java.net/jira/browse/WEBSOCKET_SPEC-211). Tomcat, Jetty, and WebSphere provide their own 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>

#### 4.2.4. Server config

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-websocket-server-config)

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>

#### 4.2.5. Allowed origins

[Same in Spring WebFlux](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web-reactive.html#webflux-websocket-server-cors)

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](https://tools.ietf.org/html/rfc6454) 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

Over the public Internet, restrictive proxies outside your control may preclude WebSocket interactions either because they are not configured to pass on the Upgrade header or because they close long lived connections that appear idle.

The solution to this problem is WebSocket emulation, i.e. attempting to use WebSocket first and then falling back on HTTP-based techniques that emulate a WebSocket interaction and expose the same application-level API.

On the Servlet stack the Spring Framework provides both server (and also client) support for the SockJS protocol.

#### 4.3.1. Overview

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](https://github.com/sockjs/sockjs-protocol) defined in the form of executable [narrated tests](https://sockjs.github.io/sockjs-protocol/sockjs-protocol-0.3.3.html).
* The [SockJS JavaScript client](https://github.com/sockjs/sockjs-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](https://github.com/sockjs/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](https://spring.io/blog/2012/05/08/spring-mvc-3-2-preview-techniques-for-real-time-updates/).

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](https://sockjs.github.io/sockjs-protocol/sockjs-protocol-0.3.3.html).

#### 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](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc-servlet). 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](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/socket/sockjs/support/SockJsHttpRequestHandler.html).

On the browser side, applications can use the [sockjs-client](https://github.com/sockjs/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](https://github.com/sockjs/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. IE 8, 9

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](https://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx). 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](https://www.owasp.org/index.php/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"](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#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](https://jira.spring.io/browse/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](https://github.com/sockjs/sockjs-client/) page. |

#### 4.3.4. Heartbeats

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](https://tools.ietf.org/html/rfc6202) 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. Client disconnects

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](https://spring.io/blog/2012/05/08/spring-mvc-3-2-preview-techniques-for-real-time-updates/).

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](https://java.net/jira/browse/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. SockJS and CORS

If you allow cross-origin requests (see [Allowed origins](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-server-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. SockJsClient

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

The WebSocket protocol defines two types of messages, text and binary, but their content is undefined. The defines a mechanism for client and server to negotiate a sub-protocol — i.e. a higher level messaging protocol, to use on top of WebSocket to define what kind of messages each can send, what is the format and content for each message, and so on. The use of a sub-protocol is optional but either way client and server will need to agree on some protocol that defines message content.

#### 4.4.1. Overview

[STOMP](https://stomp.github.io/stomp-specification-1.2.html#Abstract) 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 minimal 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, message payloads 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](https://stomp.github.io/stomp-specification-1.2.html) in full.

#### 4.4.2. Benefits

Use of STOMP as a sub-protocol enables the Spring Framework and Spring Security to provide a richer programming model vs using raw WebSockets. The same point can be made about how HTTP vs raw TCP and how it enables Spring MVC and other web frameworks to provide rich functionality. The following is a list of benefits:

* No need to invent a custom messaging protocol and message format.
* STOMP clients are available including a [Java client](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-stomp-client) in the Spring Framework.
* Message brokers such as RabbitMQ, ActiveMQ, and others can be used (optionally) to manage subscriptions and broadcast messages.
* Application logic can be organized in any number of @Controller's and messages routed to them based on the STOMP destination header vs handling raw WebSocket messages with a single WebSocketHandler for a given connection.
* Use Spring Security to secure messages based on STOMP destinations and message types.

#### 4.4.3. Enable STOMP

STOMP over WebSocket support is available in the spring-messaging and the spring-websocket modules. Once you have those dependencies, you can expose a STOMP endpoints, over WebSocket with [SockJS Fallback](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-fallback), as shown below:

**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");

}

}

|  |  |
| --- | --- |
|  | "/portfolio" is the HTTP URL for the endpoint to which a WebSocket (or SockJS) client will need to connect to for the WebSocket handshake. |
|  | STOMP messages whose destination header begins with "/app" are routed to @MessageMapping methods in @Controllerclasses. |
|  | Use the built-in, message broker for subscriptions and broadcasting; Route messages whose destination header begins with "/topic" or "/queue" to the broker. |

The same configuration 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>

|  |  |
| --- | --- |
|  | For the built-in, simple broker the "/topic" and "/queue" prefixes do not have any special meaning. They’re merely a convention to differentiate between pub-sub vs point-to-point messaging (i.e. many subscribers vs one consumer). When using an external broker, please check the STOMP page of the broker to understand what kind of STOMP destinations and prefixes it supports. |

To connect from a browser, for SockJS you can use the [sockjs-client](https://github.com/sockjs/sockjs-client). For STOMP many applications have used the [jmesnil/stomp-websocket](https://github.com/jmesnil/stomp-websocket) library (also known as stomp.js) which is feature complete and has been used in production for years but is no longer maintained. At present the [JSteunou/webstomp-client](https://github.com/JSteunou/webstomp-client) is the most actively maintained and evolving successor of that library and the example code below is based on it:

**var** socket = **new** SockJS("/spring-websocket-portfolio/portfolio");

**var** stompClient = webstomp.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 [Connect to Broker](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-stomp-handle-broker-relay-configure) and [Authentication](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-stomp-authentication) for more information on authentication.

For a more example code see:

* [Using WebSocket to build an interactive web application](https://spring.io/guides/gs/messaging-stomp-websocket/) getting started guide.
* [Stock Portfolio](https://github.com/rstoyanchev/spring-websocket-portfolio) sample application.

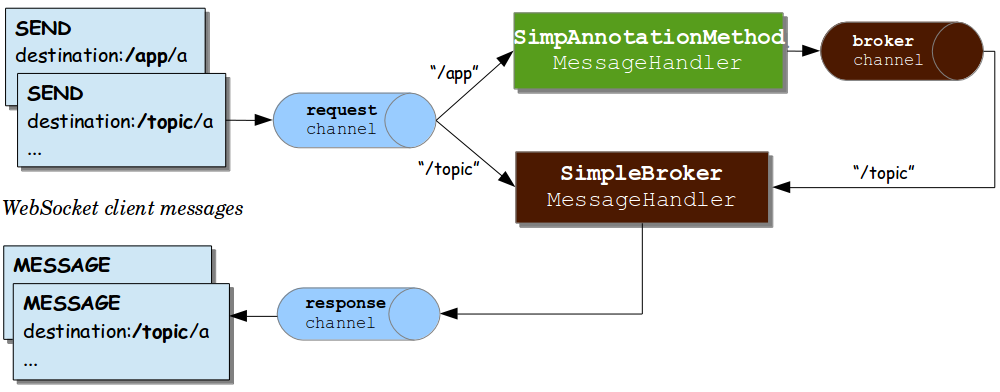
#### 4.4.4. Flow of Messages

Once a STOMP endpoint is exposed, the Spring application becomes a STOMP broker for connected clients. This section describes the flow of messages on the server side.

The spring-messaging module contains foundational support for messaging applications that originated in [Spring Integration](https://spring.io/spring-integration)and was later extracted and incorporated into the Spring Framework for broader use across many [Spring projects](https://spring.io/projects) and application scenarios. Below is a list of a few of the available messaging abstractions:

* [Message](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/messaging/Message.html) — simple representation for a message including headers and payload.
* [MessageHandler](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/messaging/MessageHandler.html) — contract for handling a message.
* [MessageChannel](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/messaging/MessageChannel.html) — contract for sending a message that enables loose coupling between producers and consumers.
* [SubscribableChannel](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/messaging/SubscribableChannel.html) — MessageChannel with MessageHandler subscribers.
* [ExecutorSubscribableChannel](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/messaging/support/ExecutorSubscribableChannel.html) — SubscribableChannel that uses an Executor for delivering messages.

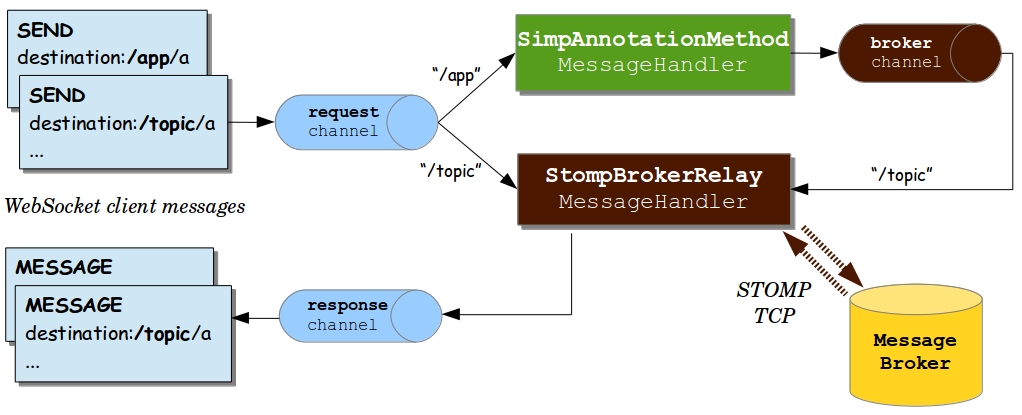
Both the Java config (i.e. @EnableWebSocketMessageBroker) and the XML namespace config (i.e. <websocket:message-broker>) use the above components to assemble a message workflow. The diagram below shows the components used when the simple, built-in message broker is enabled:



There are 3 message channels in the above diagram:

* "clientInboundChannel" — for passing messages received from WebSocket clients.
* "clientOutboundChannel" — for sending server messages to WebSocket clients.
* "brokerChannel" — for sending messages to the message broker from within server-side, application code.

The next diagram shows the components used when an external broker (e.g. RabbitMQ) is configured for managing subscriptions and broadcasting messages:



The main difference in the above diagram is the use of the "broker relay" for passing messages up to the external STOMP broker over TCP, and for passing messages down from the broker to subscribed clients.

When messages are received from a WebSocket connectin, they’re decoded to STOMP frames, then turned into a Spring Message representation, and sent to the "clientInboundChannel" for further processing. For example STOMP messages whose destination header starts with "/app" may be routed to @MessageMapping methods in annotated controllers, while "/topic"and "/queue" messages may be routed directly to the message broker.

An annotated @Controller handling a STOMP message from a client may send a message to the message broker through the "brokerChannel", and the broker will broadcast the message to matching subscribers through the "clientOutboundChannel". The same controller can also do the same in response to HTTP requests, so a client may perform an HTTP POST and then an @PostMapping method can send a message to the message broker to broadcast to subscribed clients.

Let’s trace the flow through a simple example. Given the following server setup:

@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;

}

}

1. Client connects to "http://localhost:8080/portfolio" and once a WebSocket connection is established, STOMP frames begin to flow on it.
2. Client sends SUBSCRIBE frame with destination header "/topic/greeting". Once received and decoded, the message is sent to the "clientInboundChannel", then routed to the message broker which stores the client subscription.
3. Client sends SEND frame to "/app/greeting". The "/app" prefix helps to route it to annotated controllers. After the "/app"prefix is stripped, the remaining "/greeting" part of the destination is mapped to the @MessageMapping method in GreetingController.
4. The value returned from GreetingController is turned into a Spring Message with a payload based on the return value and a default destination header of "/topic/greeting" (derived from the input destination with "/app" replaced by "/topic"). The resulting message is sent to the "brokerChannel" and handled by the message broker.
5. The message broker finds all matching subscribers, and sends a MESSAGE frame to each through the "clientOutboundChannel" from where messages are encoded as STOMP frames and sent on the WebSocket connection.

The next section provides more details on annotated methods including the kinds of arguments and return values supported.

#### 4.4.5. Handler methods

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 [Dot as Separator](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-stomp-destination-separator). |

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 aorg.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/CompletionStagereturn 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" />. |

#### 4.4.6. Send 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.

#### 4.4.7. 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 [Dot as Separator](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-stomp-destination-separator). |

#### 4.4.8. External 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](https://www.rabbitmq.com/stomp.html), [ActiveMQ](http://activemq.apache.org/stomp.html), 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](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/messaging/MessageHandler.html) 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 add 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 [Send Messages](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-stomp-handle-send), in order to broadcast messages to subscribed WebSocket clients.

In effect, the broker relay enables robust and scalable message broadcasting.

#### 4.4.9. Connect to 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 hostheader 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.

#### 4.4.10. Dot as Separator

When messages are routed to @MessageMapping methods, they’re matched with AntPathMatcher and by default patterns are expected to use slash "/" as separator. This is a good convention in a web applications and similar to HTTP URLs. However if you are more used to messaging conventions, you can switch to using dot "." as separator.

In Java config:

@Configuration

@EnableWebSocketMessageBroker

**public** **class** **WebSocketConfig** **implements** WebSocketMessageBrokerConfigurer {

*// ...*

@Override

**public** **void** configureMessageBroker(MessageBrokerRegistry registry) {

registry.setPathMatcher(**new AntPathMatcher("."));**

registry.enableStompBrokerRelay("/queue", "/topic");

registry.setApplicationDestinationPrefixes("/app");

}

}

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" path-matcher="**pathMatcher**">

<websocket:stomp-endpoint path="/stomp"/>

<websocket:stomp-broker-relay prefix="/topic,/queue" />

</websocket:message-broker>

**<bean id="pathMatcher" class="org.springframework.util.AntPathMatcher">**

**<constructor-arg index="0" value="."/>**

**</bean>**

</beans>

After that a controller may use dot "." as separator in @MessageMapping methods:

@Controller

@MessageMapping("foo")

**public** **class** **FooController** {

@MessageMapping("bar.{baz}")

**public** **void** handleBaz(@DestinationVariable String baz) {

*// ...*

}

}

The client can now send a message to "/app/foo.bar.baz123".

In the example above we did not change the prefixes on the "broker relay" because those depend entirely on the external message broker. Check the STOMP documentation pages of the broker you’re using to see what conventions it supports for the destination header.

The "simple broker" on the other hand does rely on the configured PathMatcher so if you switch the separator that will also apply to the broker and the way matches destinations from a message to patterns in subscriptions.

#### 4.4.11. 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](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#websocket) that uses a ChannelInterceptor to authorize messages based on the user header in them. Also Spring Session provides a [WebSocket integration](https://docs.spring.io/spring-session/docs/current/reference/html5/#websocket) that ensures the user HTTP session does not expire when the WebSocket session is still active. |

#### 4.4.12. Token Authentication

[Spring Security OAuth](https://github.com/spring-projects/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](https://tools.ietf.org/html/rfc6455#section-10.5) "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](https://github.com/sockjs/sockjs-client/issues/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** **implements** WebSocketMessageBrokerConfigurer {

@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 implementation of WebSocketMessageBrokerConfigurer marked with @Order(Ordered.HIGHEST\_PRECEDENCE + 99).

#### 4.4.13. 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 @SendToUserannotation 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](http://activemq.apache.org/delete-inactive-destinations.html) 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.

#### 4.4.14. Events and Interception

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 ChannelInterceptoron the respective message channel. For example to intercept inbound messages:

@Configuration

@EnableWebSocketMessageBroker

**public** **class** **WebSocketConfig** **implements** WebSocketMessageBrokerConfigurer {

@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;

}

}

#### 4.4.15. 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[SockJsClient](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-fallback-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.

#### 4.4.16. 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.

#### 4.4.17. 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](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket-stomp-message-flow).

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 the JavaScript [webstomp-client](https://github.com/JSteunou/webstomp-client) and others 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.

#### 4.4.18. 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.

#### 4.4.19. Testing

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](https://github.com/rstoyanchev/spring-websocket-portfolio/tree/master/src/test/java/org/springframework/samples/portfolio/web) 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](https://github.com/rstoyanchev/spring-websocket-portfolio/tree/master/src/test/java/org/springframework/samples/portfolio/web) sample application also demonstrates this approach using Tomcat as the embedded WebSocket server and a simple STOMP client for test purposes.

## 5. Other Web Frameworks

### 5.1. Introduction

This chapter details Spring’s integration with third party web frameworks.

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](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc)), while at the same time providing integration with a number of popular third party web frameworks.

### 5.2. Common config

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](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/context/ContextLoaderListener.html) 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](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/context/WebApplicationContext.html) 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](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/context/support/WebApplicationContextUtils.html) class is for convenience, so you don’t have to remember the name of the ServletContextattribute. 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.

### 5.3. JSF

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](https://myfaces.apache.org/). The MyFaces project also provides common JSF extensions such as [MyFaces Orchestra](https://myfaces.apache.org/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](https://projects.spring.io/spring-webflow) for details! |

The key element in Spring’s JSF integration is the JSF ELResolver mechanism.

#### 5.3.1. Spring Bean Resolver

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 faces-context.xml file:

<faces-config>

<application>

<el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>

...

</application>

</faces-config>

#### 5.3.2. 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](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/jsf/FacesContextUtils.html) 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());

### 5.4. Apache Struts 2.x

Invented by Craig McClanahan, [Struts](https://struts.apache.org/) 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](https://struts.apache.org/release/2.3.x/docs/spring-plugin.html) for the built-in Spring integration shipped with Struts.

### 5.5. Tapestry 5.x

From the [Tapestry homepage](https://tapestry.apache.org/):

Tapestry is a "Component oriented framework for creating dynamic, robust, highly scalable web applications in Java."

While Spring has its own [powerful web layer](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#mvc), 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](https://tapestry.apache.org/integrating-with-spring-framework.html).

### 5.6. Further Resources

Find below links to further resources about the various web frameworks described in this chapter.

* The [JSF](http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html) homepage
* The [Struts](https://struts.apache.org/) homepage
* The [Tapestry](https://tapestry.apache.org/) homepage

Version 5.0.3.RELEASE  
Last updated 2018-01-23 09:11:25 UTC