Servlet Web Applications

If you want to build servlet-based web applications, you can take advantage of Spring Boot's auto-configuration for Spring MVC or Jersey.

The "Spring Web MVC Framework"

The Spring Web MVC framework (often referred to as "Spring MVC") is a rich "model view controller" web framework. Spring MVC lets you create special `@Controller` or `@RestController` beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using `@RequestMapping` annotations.

The following code shows a typical `@RestController` that serves JSON data:


```java
@RequestMapping("/users")
public class MyRestController {

    private final UserRepository userRepository;

    private final CustomerRepository customerRepository;

    public MyRestController(UserRepository userRepository, CustomerRepository customerRepository) {
        this.userRepository = userRepository;
        this.customerRepository = customerRepository;
    }

    @GetMapping("/{userId}")
    public User getUser(@PathVariable Long userId) {
        return this.userRepository.findById(userId).get();
    }

    @GetMapping("/{userId}/customers")
    public List<Customer> getUserCustomers(@PathVariable Long userId) {
        return this.userRepository.findById(userId).map(this.customerRepository::findByUser).get();
    }

    @DeleteMapping("/{userId}")
    public void deleteUser(@PathVariable Long userId) {
        this.userRepository.deleteById(userId);
    }

}
```

"WebMvc.fn", the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

Java Kotlin

```java
@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

    private static final RequestPredicate ACCEPT_JSON = accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> routerFunction(MyUserHandler userHandler) {
        return route()
                .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
                .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
                .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
                .build();
    }

}
```

Java Kotlin

```java
@Component
public class MyUserHandler {

    public ServerResponse getUser(ServerRequest request) {
        ...
    }

    public ServerResponse getUserCustomers(ServerRequest request) {
        ...
    }

    public ServerResponse deleteUser(ServerRequest request) {
        ...
    }

}
```

Spring MVC is part of the core Spring Framework, and detailed information is available in the reference documentation. There are also several guides that cover Spring MVC available at spring.io/guides.

TIP

You can define as many `RouterFunction` beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

Spring MVC Auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications. It replaces the need for `@EnableWebMvc` and the two cannot be used together. In addition to Spring MVC's defaults, the auto-configuration provides the following features:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered later in this document).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.

- Support for `HttpMessageConverters` (covered later in this document).
- Automatic registration of `MessageCodesResolver` (covered later in this document).
- Static `index.html` support.
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).

If you want to keep those Spring Boot MVC customizations and make more MVC customizations (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`.

If you want to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, and still keep the Spring Boot MVC customizations, you can declare a bean of type `WebMvcRegistrations` and use it to provide custom instances of those components. The custom instances will be subject to further initialization and configuration by Spring MVC. To participate in, and if desired, override that subsequent processing, a `WebMvcConfigurer` should be used.

If you do not want to use the auto-configuration and want to take complete control of Spring MVC, add your own `@Configuration` annotated with `@EnableWebMvc`. Alternatively, add your own `@Configuration`-annotated `DelegatingWebMvcConfiguration` as described in the `@EnableWebMvc` API documentation.

Spring MVC Conversion Service

Spring MVC uses a different `ConversionService` to the one used to convert values from your `application.properties` or `application.yaml` file. It means that `Period`, `Duration` and `DataSize` converters are not available and that `@DurationUnit` and `@DataSizeUnit` annotations will be ignored.

If you want to customize the `ConversionService` used by Spring MVC, you can provide a `WebMvcConfigurer` bean with an `addFormatters` method. From this method you can register any converter that you like, or you can delegate to the static methods available on `ApplicationConversionService`.

Conversion can also be customized using the `spring.mvc.format.*` configuration properties. When not configured, the following defaults are used:

Property	`DateTimeFormatter`	Formats
`spring.mvc.format.date`	`ofLocalizedDate(FormatStyle.SHORT)`	`java.util.Date` and `LocalDate`
`spring.mvc.format.time`	`ofLocalizedTime(FormatStyle.SHORT)`	java.time's `LocalTime` and `OffsetTime`
`spring.mvc.format.date-time`	`ofLocalizedDateTime(FormatStyle.SHORT)`	java.time's `LocalDateTime`, `OffsetDateTime`, and `ZonedDateTime`

HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in `UTF-8`.

Any `HttpMessageConverter` bean that is present in the context is added to the list of converters. You can also override default converters in the same way.

If you need to add or customize converters, you can use Spring Boot's `HttpMessageConverters` class, as shown in the following listing:

Java | Kotlin

```java
@Configuration(proxyBeanMethods = false)
public class MyHttpMessageConvertersConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = new AdditionalHttpMessageConverter();
        HttpMessageConverter<?> another = new AnotherHttpMessageConverter();
        return new HttpMessageConverters(additional, another);
    }

}
```

For further control, you can also sub-class `HttpMessageConverters` and override its `postProcessConverters` and/or `postProcessPartConverters` methods. This can be useful when you want to re-order or remove some of the converters that Spring MVC configures by default.

MessageCodesResolver

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: `MessageCodesResolver`. If you set the `spring.mvc.message-codes-resolver-format` property `PREFIX_ERROR_CODE` or `POSTFIX_ERROR_CODE`, Spring Boot creates one for you (see the enumeration in `DefaultMessageCodesResolver.Format`).

Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServletContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so that you can modify that behavior by adding your own `WebMvcConfigurer` and overriding the `addResourceHandlers` method.

In a stand-alone web application, the default servlet from the container is not enabled. It can be enabled using the `server.servlet.register-default-servlet` property.

The default servlet acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through the `DispatcherServlet`.

By default, resources are mapped on `/**`, but you can tune that with the `spring.mvc.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

Properties | YAML

```properties
spring.mvc.static-path-pattern=/resources/**
```

You can also customize the static resource locations by using the `spring.web.resources.static-locations` property (replacing the default values with a list of directory locations). The root servlet context path, `"/"`, is automatically added as a location as well.

In addition to the "standard" static resource locations mentioned earlier, a special case is made for Webjars content. By default, any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format. The path can be customized with the `spring.mvc.webjars-path-pattern` property.

TIP

Do not use the `src/main/webapp` directory if your application is packaged as a jar. Although this directory is a common standard, it works **only** with war packaging, and it is silently ignored by most build tools if you generate a jar.

Spring Boot also supports the advanced resource handling features provided by Spring MVC, allowing use cases such as cache-busting static resources or using version agnostic URLs for Webjars.

To use version agnostic URLs for Webjars, add the `org.webjars:webjars-locator-lite` dependency. Then declare your Webjar. Using jQuery as an example, adding `"/webjars/jquery/jquery.min.js"` results in `"/webjars/jquery/x.y.z/jquery.min.js"` where `x.y.z` is the Webjar version.

To use cache busting, the following configuration configures a cache busting solution for all static resources, effectively adding a content hash, such as `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>` , in URLs:

Properties | YAML

```
PROPERTIES
spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.content.paths=/**
```

NOTE

Links to resources are rewritten in templates at runtime, thanks to a `ResourceUrlEncodingFilter` that is auto-configured for Thymeleaf and FreeMarker. You should manually declare this filter when using JSPs. Other template engines are currently not automatically supported but can be with custom template macros/helpers and the use of the `ResourceUrlProvider` .

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That is why other strategies are also supported and can be combined. A "fixed" strategy adds a static version string in the URL without changing the file name, as shown in the following example:

Properties | YAML

```
PROPERTIES
spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.content.paths=/**
spring.web.resources.chain.strategy.fixed.enabled=true
spring.web.resources.chain.strategy.fixed.paths=/js/lib/
spring.web.resources.chain.strategy.fixed.version=v12
```

With this configuration, JavaScript modules located under `"/js/lib/"` use a fixed versioning strategy (`"/v12/js/lib/mymodule.js"`), while other resources still use the content one (`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`).

See `WebProperties.Resources` for more supported options.

TIP

This feature has been thoroughly described in a dedicated blog post and in Spring Framework's reference documentation.

Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

This only acts as a fallback for actual index routes defined by the application. The ordering is defined by the order of `HandlerMapping` beans which is by default the following:

`RouterFunctionMapping`	Endpoints declared with `RouterFunction` beans
`RequestMappingHandlerMapping`	Endpoints declared in `@Controller` beans
`WelcomePageHandlerMapping`	The welcome page support

Custom Favicon

As with other static resources, Spring Boot checks for a `favicon.ico` in the configured static content locations. If such a file is present, it is automatically used as the favicon of the application.

Path Matching and Content Negotiation

Spring MVC can map incoming HTTP requests to handlers by looking at the request path and matching it to the mappings defined in your application (for example, `@GetMapping` annotations on Controller methods).

Spring Boot chooses to disable suffix pattern matching by default, which means that requests like `"GET /projects/spring-boot.json"` will not be matched to `@GetMapping("/projects/spring-boot")` mappings. This is considered as a best practice for Spring MVC applications. This feature was mainly useful in the past for HTTP clients which did not send proper "Accept" request headers; we needed to make sure to send the correct Content Type to the client. Nowadays, Content Negotiation is much more reliable.

There are other ways to deal with HTTP clients that do not consistently send proper "Accept" request headers. Instead of using suffix matching, we can use a query parameter to ensure that requests like `"GET /projects/spring-boot?format=json"` will be mapped to `@GetMapping("/projects/spring-boot")` :

Properties | YAML

```
PROPERTIES
spring.mvc.contentnegotiation.favor-parameter=true
```

Or if you prefer to use a different parameter name:

Properties | YAML

```
PROPERTIES
spring.mvc.contentnegotiation.favor-parameter=true
spring.mvc.contentnegotiation.parameter-name=myparam
```

Most standard media types are supported out-of-the-box, but you can also define new ones:

Properties | YAML

```
PROPERTIES
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

As of Spring Framework 5.3, Spring MVC supports two strategies for matching request paths to controllers. By default, Spring Boot uses the `PathPatternParser` strategy. `PathPatternParser` is an optimized implementation but comes with some restrictions compared to the `AntPathMatcher` strategy. `PathPatternParser` restricts usage of some path pattern variants. It is also incompatible with configuring the `DispatcherServlet` with a path prefix (`spring.mvc.servlet.path`).

The strategy can be configured using the `spring.mvc.pathmatch.matching-strategy` configuration property, as shown in the following example:

Properties | YAML

```
PROPERTIES
spring.mvc.pathmatch.matching-strategy=ant-path-matcher
```

Spring MVC will throw a `NoHandlerFoundException` if a handler is not found for a request. Note that, by default, the serving of static content is mapped to `/**` and will, therefore, provide a handler for all requests. If no static content is available, `ResourceHttpRequestHandler` will throw a `NoResourceFoundException` . For a `NoHandlerFoundException` to be thrown, set `spring.mvc.static-path-pattern` to a more specific value

such as `/resources/**` or set `spring.web.resources.add-mappings` to `false` to disable serving of static content entirely.

ConfigurableWebBindingInitializer

Spring MVC uses a `WebBindingInitializer` to initialize a `WebDataBinder` for a particular request. If you create your own `ConfigurableWebBindingInitializer` `@Bean`, Spring Boot automatically configures Spring MVC to use it.

Template Engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

TIP

If possible, JSPs should be avoided. There are several known limitations when using them with embedded servlet containers.

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

TIP

Depending on how you run your application, your IDE may order the classpath differently. Running your application in the IDE from its main method results in a different ordering than when you run your application by using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the expected template. If you have this problem, you can reorder the classpath in the IDE to place the module's classes and resources first.

Error Handling

By default, Spring Boot provides an `/error` mapping that handles all errors in a sensible way, and it is registered as a "global" error page in the servlet container. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a "whitelabel" error view that renders the same data in HTML format (to customize it, add a `View` that resolves to `error`).

There are a number of `server.error` properties that can be set if you want to customize the default error handling behavior. See the Server Properties section of the Appendix.

To replace the default behavior completely, you can implement `ErrorController` and register a bean definition of that type or add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.

TIP

The `BasicErrorController` can be used as a base class for a custom `ErrorController`. This is particularly useful if you want to add a handler for a new content type (the default is to handle `text/html` specifically and provide a fallback for everything else). To do so, extend `BasicErrorController`, add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

As of Spring Framework 6.0, RFC 9457 Problem Details is supported. Spring MVC can produce custom error messages with the `application/problem+json` media type, like:

JSON

```
{
    "type": "https://example.org/problems/unknown-project",
    "title": "Unknown project",
    "status": 404,
    "detail": "No project found for id 'spring-unknown'",
    "instance": "/projects/spring-unknown"
}
```

This support can be enabled by setting `spring.mvc.problemdetails.enabled` to `true`.

You can also define a class annotated with `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type, as shown in the following example:

Java Kotlin

JAVA

```
@ControllerAdvice(basePackageClasses = SomeController.class)
public class MyControllerAdvice extends ResponseEntityExceptionHandler {

    @ResponseBody
    @ExceptionHandler(MyException.class)
    public ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {
        HttpStatus status = getStatus(request);
        return new ResponseEntity<>(new MyErrorBody(status.value(), ex.getMessage()), status);
    }

    private HttpStatus getStatus(HttpServletRequest request) {
        Integer code = (Integer) request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
        HttpStatus status = HttpStatus.resolve(code);
        return (status != null) ? status : HttpStatus.INTERNAL_SERVER_ERROR;
    }

}
```

In the preceding example, if `MyException` is thrown by a controller defined in the same package as `SomeController`, a JSON representation of the `MyErrorBody` POJO is used instead of the `ErrorAttributes` representation.

In some cases, errors handled at the controller level are not recorded by web observations or the metrics infrastructure. Applications can ensure that such exceptions are recorded with the observations by setting the handled exception on the observation context.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` directory. Error pages can either be static HTML (that is, added under any of the static resource directories) or be built by using templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your directory structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
```

```
            |   + 404.html
            +- <other public assets>
```

To map all `5xx` errors by using a FreeMarker template, your directory structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.ftlh
             +- <other templates>
```

For more complex mappings, you can also add beans that implement the `ErrorViewResolver` interface, as shown in the following example:

Java Kotlin

```
public class MyErrorViewResolver implements ErrorViewResolver {

    @Override
    public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String, Object> model) {
        // Use the request or status to optionally return a ModelAndView
        if (status == HttpStatus.INSUFFICIENT_STORAGE) {
            // We could add custom model values here
            new ModelAndView("myview");
        }
        return null;
    }

}
```

You can also use regular Spring MVC features such as `@ExceptionHandler` methods and `@ControllerAdvice`. The `ErrorController` then picks up any unhandled exceptions.

Mapping Error Pages Outside of Spring MVC

For applications that do not use Spring MVC, you can use the `ErrorPageRegistrar` interface to directly register `ErrorPage` instances. This abstraction works directly with the underlying embedded servlet container and works even if you do not have a Spring MVC `DispatcherServlet`.

Java Kotlin

```
@Configuration(proxyBeanMethods = false)
public class MyErrorPagesConfiguration {

    @Bean
    public ErrorPageRegistrar errorPageRegistrar() {
        return this::registerErrorPages;
    }

    private void registerErrorPages(ErrorPageRegistry registry) {
        registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }

}
```

NOTE

If you register an `ErrorPage` with a path that ends up being handled by a `Filter` (as is common with some non-Spring web frameworks, like Jersey and Wicket), then the `Filter` has to be explicitly registered as an `ERROR` dispatcher, as shown in the following example:

Java Kotlin

```
@Configuration(proxyBeanMethods = false)
public class MyFilterConfiguration {

    @Bean
    public FilterRegistrationBean<MyFilter> myFilter() {
        FilterRegistrationBean<MyFilter> registration = new FilterRegistrationBean<>(new MyFilter());
        // ...
        registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class));
        return registration;
    }

}
```

Note that the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type.

Error Handling in a WAR Deployment

When deployed to a servlet container, Spring Boot uses its error page filter to forward a request with an error status to the appropriate error page. This is necessary as the servlet specification does not provide an API for registering error pages. Depending on the container that you are deploying your war file to and the technologies that your application uses, some additional configuration may be required.

The error page filter can only forward the request to the correct error page if the response has not already been committed. By default, WebSphere Application Server 8.0 and later commits the response upon successful completion of a servlet's service method. You should disable this behavior by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`.

CORS Support

Cross-origin resource sharing (CORS) is a W3C specification implemented by most browsers that lets you specify in a flexible way what kind of cross-domain requests are authorized, instead of using some less secure and less powerful approaches such as IFRAME or JSONP.

As of version 4.2, Spring MVC supports CORS. Using controller method CORS configuration with `@CrossOrigin` annotations in your Spring Boot application does not require any specific configuration. Global CORS configuration can be defined by registering a `WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method, as shown in the following example:

Java Kotlin

```
@Configuration(proxyBeanMethods = false)
public class MyCorsConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {

            @Override
            public void addCorsMappings(CorsRegistry registry) {
```

```
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**");
            }

        };
    }

}
```

JAX-RS and Jersey

If you prefer the JAX-RS programming model for REST endpoints, you can use one of the available implementations instead of Spring MVC. Jersey and Apache CXF work quite well out of the box. CXF requires you to register its `Servlet` or `Filter` as a `@Bean` in your application context. Jersey has some native Spring support, so we also provide auto-configuration support for it in Spring Boot, together with a starter.

To get started with Jersey, include the `spring-boot-starter-jersey` as a dependency and then you need one `@Bean` of type `ResourceConfig` in which you register all the endpoints, as shown in the following example:

```java
@Component
public class MyJerseyConfig extends ResourceConfig {

    public MyJerseyConfig() {
        register(MyEndpoint.class);
    }

}
```

> **WARNING**
>
> Jersey's support for scanning executable archives is rather limited. For example, it cannot scan for endpoints in a package found in a fully executable jar file or in `WEB-INF/classes` when running an executable war file. To avoid this limitation, the `packages` method should not be used, and endpoints should be registered individually by using the `register` method, as shown in the preceding example.

For more advanced customizations, you can also register an arbitrary number of beans that implement `ResourceConfigCustomizer`.

All the registered endpoints should be a `@Component` with HTTP resource annotations (`@GET` and others), as shown in the following example:

```java
@Component
@Path("/hello")
public class MyEndpoint {

    @GET
    public String message() {
        return "Hello";
    }

}
```

Since the `@Endpoint` is a Spring `@Component`, its lifecycle is managed by Spring and you can use the `@Autowired` annotation to inject dependencies and use the `@Value` annotation to inject external configuration. By default, the Jersey servlet is registered and mapped to `/*`. You can change the mapping by adding `@ApplicationPath` to your `ResourceConfig`.

By default, Jersey is set up as a servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. By default, the servlet is initialized lazily, but you can customize that behavior by setting `spring.jersey.servlet.load-on-startup`. You can disable or override that bean by creating one of your own with the same name. You can also use a filter instead of a servlet by setting `spring.jersey.type=filter` (in which case, the `@Bean` to replace or override is `jerseyFilterRegistration`). The filter has an `@Order`, which you can set with `spring.jersey.filter.order`. When using Jersey as a filter, a servlet that will handle any requests that are not intercepted by Jersey must be present. If your application does not contain such a servlet, you may want to enable the default servlet by setting `server.servlet.register-default-servlet` to `true`. Both the servlet and the filter registrations can be given init parameters by using `spring.jersey.init.*` to specify a map of properties.

Embedded Servlet Container Support

For servlet application, Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. Most developers use the appropriate starter to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port `8080`.

Servlets, Filters, and Listeners

When using an embedded servlet container, you can register servlets, filters, and all the listeners (such as `HttpSessionListener`) from the servlet spec, either by using Spring beans or by scanning for servlet components.

Registering Servlets, Filters, and Listeners as Spring Beans

Any `Servlet`, `Filter`, or servlet `*Listener` instance that is a Spring bean is registered with the embedded container. This can be particularly convenient if you want to refer to a value from your `application.properties` during configuration.

By default, if the context contains only a single Servlet, it is mapped to `/`. In the case of multiple servlet beans, the bean name is used as a path prefix. Filters map to `/*`.

If convention-based mapping is not flexible enough, you can use the `ServletRegistrationBean`, `FilterRegistrationBean`, and `ServletListenerRegistrationBean` classes for complete control. If you prefer annotations over `ServletRegistrationBean` and `FilterRegistrationBean`, you can also use `@ServletRegistration` and `@FilterRegistration` as an alternative.

It is usually safe to leave filter beans unordered. If a specific order is required, you should annotate the `Filter` with `@Order` or make it implement `Ordered`. You cannot configure the order of a `Filter` by annotating its bean method with `@Order`. If you cannot change the `Filter` class to add `@Order` or implement `Ordered`, you must define a `FilterRegistrationBean` for the `Filter` and set the registration bean's order using the `setOrder(int)` method. Or, if you prefer annotations, you can also use `@FilterRegistration` and set the `order` attribute. Avoid configuring a filter that reads the request body at `Ordered.HIGHEST_PRECEDENCE`, since it might go against the character encoding configuration of your application. If a servlet filter wraps the request, it should be configured with an order that is less than or equal to `OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER`.

> **TIP**
>
> To see the order of every `Filter` in your application, enable debug level logging for the `web` logging group (`logging.level.web=debug`). Details of the registered filters, including their order and URL patterns, will then be logged at startup.

> **WARNING**
>
> Take care when registering `Filter` beans since they are initialized very early in the application lifecycle. If you need to register a `Filter` that interacts with other beans, consider using a `DelegatingFilterProxyRegistrationBean` instead.

Servlet Context Initialization

Embedded servlet containers do not directly execute the `ServletContainerInitializer` interface or Spring's `WebApplicationInitializer` inter-

face. This is an intentional design decision intended to reduce the risk that third party libraries designed to run inside a war may break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the `ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext` and, if necessary, can easily be used as an adapter to an existing `WebApplicationInitializer`.

Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of classes annotated with `@WebServlet`, `@WebFilter`, and `@WebListener` can be enabled by using `@ServletComponentScan`.

TIP

`@ServletComponentScan` has no effect in a standalone container, where the container's built-in discovery mechanisms are used instead.

The ServletWebServerApplicationContext

Under the hood, Spring Boot uses a different type of `ApplicationContext` for embedded servlet container support. The `ServletWebServerApplicationContext` is a special type of `WebApplicationContext` that bootstraps itself by searching for a single `ServletWebServerFactory` bean. Usually a `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` has been auto-configured.

NOTE

You usually do not need to be aware of these implementation classes. Most applications are auto-configured, and the appropriate `ApplicationContext` and `ServletWebServerFactory` are created on your behalf.

In an embedded container setup, the `ServletContext` is set as part of server startup which happens during application context initialization. Because of this beans in the `ApplicationContext` cannot be reliably initialized with a `ServletContext`. One way to get around this is to inject `ApplicationContext` as a dependency of the bean and access the `ServletContext` only when it is needed. Another way is to use a callback once the server has started. This can be done using an `ApplicationListener` which listens for the `ApplicationStartedEvent` as follows:

JAVA

```
public class MyDemoBean implements ApplicationListener<ApplicationStartedEvent> {

    private ServletContext servletContext;

    @Override
    public void onApplicationEvent(ApplicationStartedEvent event) {
        ApplicationContext applicationContext = event.getApplicationContext();
        this.servletContext = ((WebApplicationContext) applicationContext).getServletContext();
    }

}
```

Customizing Embedded Servlet Containers

Common servlet container settings can be configured by using Spring `Environment` properties. Usually, you would define the properties in your `application.properties` or `application.yaml` file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to (`server.address`), and so on.
- Session settings: Whether the session is persistent (`server.servlet.session.persistent`), session timeout (`server.servlet.session.timeout`), location of session data (`server.servlet.session.store-dir`), and session-cookie configuration (`server.servlet.session.cookie.*`).
- Error management: Location of the error page (`server.error.path`) and so on.
- SSL
- HTTP compression

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see `server.tomcat` and `server.undertow`). For instance, access logs can be configured with specific features of the embedded servlet container.

TIP

See the `ServerProperties` class for a complete list.

SameSite Cookies

The `SameSite` cookie attribute can be used by web browsers to control if and how cookies are submitted in cross-site requests. The attribute is particularly relevant for modern web browsers which have started to change the default value that is used when the attribute is missing.

If you want to change the `SameSite` attribute of your session cookie, you can use the `server.servlet.session.cookie.same-site` property. This property is supported by auto-configured Tomcat, Jetty and Undertow servers. It is also used to configure Spring Session servlet based `SessionRepository` beans.

For example, if you want your session cookie to have a `SameSite` attribute of `None`, you can add the following to your `application.properties` or `application.yaml` file:

Properties | YAML

PROPERTIES

```
server.servlet.session.cookie.same-site=none
```

If you want to change the `SameSite` attribute on other cookies added to your `HttpServletResponse`, you can use a `CookieSameSiteSupplier`. The `CookieSameSiteSupplier` is passed a `Cookie` and may return a `SameSite` value, or `null`.

There are a number of convenience factory and filter methods that you can use to quickly match specific cookies. For example, adding the following bean will automatically apply a `SameSite` of `Lax` for all cookies with a name that matches the regular expression `myapp.*`.

Java | Kotlin

JAVA

```
@Configuration(proxyBeanMethods = false)
public class MySameSiteConfiguration {

    @Bean
    public CookieSameSiteSupplier applicationCookieSameSiteSupplier() {
        return CookieSameSiteSupplier.ofLax().whenHasNameMatching("myapp.*");
    }

}
```

Character Encoding

The character encoding behavior of the embedded servlet container for request and response handling can be configured using the

The character encoding behavior of the embedded servlet container for request and response handling can be configured using the `server.servlet.encoding.*` configuration properties.

When a request's `Accept-Language` header indicates a locale for the request it will be automatically mapped to a charset by the servlet container. Each container provides default locale to charset mappings and you should verify that they meet your application's needs. When they do not, use the `server.servlet.encoding.mapping` configuration property to customize the mappings, as shown in the following example:

Properties | YAML

```properties
server.servlet.encoding.mapping.ko=UTF-8
```

In the preceding example, the `ko` (Korean) locale has been mapped to `UTF-8`. This is equivalent to a `<locale-encoding-mapping-list>` entry in a `web.xml` file of a traditional war deployment.

Programmatic Customization

If you need to programmatically configure your embedded servlet container, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableServletWebServerFactory`, which includes numerous customization setter methods. The following example shows programmatically setting the port:

Java | Kotlin

```java
@Component
public class MyWebServerFactoryCustomizer implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory server) {
        server.setPort(9000);
    }

}
```

`TomcatServletWebServerFactory`, `JettyServletWebServerFactory` and `UndertowServletWebServerFactory` are dedicated variants of `ConfigurableServletWebServerFactory` that have additional customization setter methods for Tomcat, Jetty and Undertow respectively. The following example shows how to customize `TomcatServletWebServerFactory` that provides access to Tomcat-specific configuration options:

Java | Kotlin

```java
@Component
public class MyTomcatWebServerFactoryCustomizer implements WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory server) {
        server.addConnectorCustomizers((connector) -> connector.setAsyncTimeout(Duration.ofSeconds(20).toMillis()));
    }

}
```

Customizing ConfigurableServletWebServerFactory Directly

For more advanced use cases that require you to extend from `ServletWebServerFactory`, you can expose a bean of such type yourself.

Setters are provided for many configuration options. Several protected method "hooks" are also provided should you need to do something more exotic. See the `ConfigurableServletWebServerFactory` API documentation for details.

NOTE

Auto-configured customizers are still applied on your custom factory, so use that option carefully.

JSP Limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

- With Jetty and Tomcat, it should work if you use war packaging. An executable war will work when launched with `java -jar`, and will also be deployable to any standard container. JSPs are not supported when using an executable jar.
- Undertow does not support JSPs.
- Creating a custom `error.jsp` page does not override the default view for error handling. Custom error pages should be used instead.