27. Developing web applications

Spring Boot is well suited for web application development. You can easily create a self-contained HTTP server using embedded Tomcat, Jetty, or Undertow. Most web applications will use the `spring-boot-starter-web` module to get up and running quickly.

If you haven't yet developed a Spring Boot web application you can follow the "Hello World!" example in the *Getting started* section.

27.1 The 'Spring Web MVC framework'

The Spring Web MVC framework (often referred to as simply '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 using `@RequestMapping` annotations.

Here is a typical example `@RestController` to serve JSON data:

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }
```

```
}
```

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

27.1.1 Spring MVC auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (see below).
- Automatic registration of `Converter`, `GenericConverter`, `Formatter` beans.
- Support for `HttpMessageConverters` (see below).
- Automatic registration of `MessageCodesResolver` (see below).
- Static `index.html` support.
- Custom `Favicon` support.
- Automatic use of a `ConfigurableWebBindingInitializer` bean (see below).

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`. If you want to keep Spring Boot MVC features, and you just want to add additional MVC configuration (interceptors, formatters, view controllers etc.) you can add your own `@Bean` of type `WebMvcConfigurerAdapter`, but **without** `@EnableWebMvc`.

27.1.2 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 (using the Jackson library) or XML (using the Jackson XML extension if available, else using JAXB). Strings are encoded using `UTF-8` by default.

If you need to add or customize converters you can use Spring Boot's `HttpMessageConverters` class:

```
import org.springframework.boot.autoconfigure.web.HttpMessageConverters;
import org.springframework.http.converter.*;
```

```
@Configuration
public class MyConfiguration {

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

}
```

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

27.1.3 MessageCodesResolver

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

27.1.4 Static Content

By default Spring Boot will serve 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 you can modify that behavior by adding your own `WebMvcConfigurerAdapter` and overriding the `addResourceHandlers` method.

In a stand-alone web application the default servlet from the container is also enabled, and acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time this will not happen (unless you modify the default MVC configuration) because Spring will always be able to handle requests through the `DispatcherServlet`.

You can customize the static resource locations using `spring.resources.staticLocations`

page detection will switch to your custom locations, so if there is an `index.html` in any of your locations on startup, it will be the home page of the application.

In addition to the 'standard' static resource locations above, a special case is made for Webjars content. Any resources with a path in `/webjars/**` will be served from jar files if they are packaged in the Webjars format.

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

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

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

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

Links to resources are rewritten at runtime in template, thanks to a `ResourceUrlEncodingFilter`, auto-configured for Thymeleaf and Velocity. You should manually declare this filter when using JSPs. Other template engines aren't automatically supported right now, 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's why other strategies are also supported and can be combined. A "fixed" strategy will add a static version string in the URL, without changing the file name:

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


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

See `ResourceProperties` for more of the supported options.

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

27.1.5 ConfigurableWebBindingInitializer

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

27.1.6 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 Velocity, FreeMarker and JSPs. Many other templating engines also ship their own Spring MVC integrations.

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

- FreeMarker
- Groovy
- Thymeleaf
- Velocity
- Mustache

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

When you're using one of these templating engines with the default configuration, your templates will be picked up automatically from `src/main/resources/templates`.

IntelliJ IDEA orders the classpath differently depending on how you run your application. Running your application in the IDE via its main method will result in a

its packaged jar. This can cause Spring Boot to fail to find the templates on the classpath. If you're affected by this problem you can reorder the classpath in the

> IDE to place the module's classes and resources first. Alternatively, you can configure the template prefix to search every templates directory on the classpath: `classpath*:/templates/`.

27.1.7 Error Handling

Spring Boot provides an `/error` mapping by default 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 will produce 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 just add a `View` that resolves to 'error'). To replace the default behaviour completely you can implement `ErrorController` and register a bean definition of that type, or simply add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.

> 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 that just extend `BasicErrorController` and add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

You can also define a `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type.

```
@ControllerAdvice(basePackageClasses = FooController.class)
public class FooControllerAdvice extends ResponseEntityExceptionHandler {

    @ExceptionHandler(YourException.class)
    @ResponseBody
    ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwa
        HttpStatus status = getStatus(request);
        return new ResponseEntity<>(new CustomErrorType(status.value(), ex.getMes
    }

    private HttpStatus getStatus(HttpServletRequest request) {
        Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.
        if (statusCode == null) {
            return HttpStatus.INTERNAL_SERVER_ERROR;
```

```
        }
        return HttpStatus.valueOf(statusCode);
    }

}
```

In the example above, if `YourException` is thrown by a controller defined in the same package as `FooController`, a json representation of the `CustomerErrorType` POJO will be used instead of the `ErrorAttributes` representation.

If you want more specific error pages for some conditions, the embedded servlet containers support a uniform Java DSL for customizing the error handling. Assuming that you have a mapping for `/400`:

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer(){
    return new MyCustomizer();
}

// ...

private static class MyCustomizer implements EmbeddedServletContainerCustomizer {

    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }

}
```

You can also use regular Spring MVC features like `@ExceptionHandler` methods and `@ControllerAdvice`. The `ErrorController` will then pick up any unhandled exceptions.

N.B. if you register an `ErrorPage` with a path that will end up being handled by a `Filter` (e.g. 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, e.g.

```
@Bean
public FilterRegistrationBean myFilter() {
    registration.setFilter(new MyFilter());
    ...
    registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class));
```

```
    return registration;
}
```

(the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type).

Error Handling on WebSphere Application Server

When deployed to a servlet container, a Spring Boot uses its error page filter to forward a request with an error status to the appropriate error page. The request can only be forwarded 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 behaviour by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`

27.1.8 Spring HATEOAS

If you're developing a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building hypermedia-based applications including a `LinkDiscoverers` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation. The `ObjectMapper` will be customized based on the `spring.jackson.*` properties or a `Jackson2ObjectMapperBuilder` bean if one exists.

You can take control of Spring HATEOAS's configuration by using `@EnableHypermediaSupport`. Note that this will disable the `ObjectMapper` customization described above.

27.1.9 CORS support

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

As of version 4.2, Spring MVC supports CORS out of the box. Using controller method CORS configuration with `@CrossOrigin` annotations in your Spring Boot application does not require

`WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method:

```
@Configuration
```

```
public class MyConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurerAdapter() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**");
            }
        };
    }
}
```

27.2 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 1.x and Apache CXF work quite well out of the box if you just register their `Servlet` or `Filter` as a `@Bean` in your application context. Jersey 2.x 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 2.x just 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:

```
@Component
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        register(Endpoint.class);
    }

}
```

All the registered endpoints should be `@Components` with HTTP resource annotations (`@GET` etc.), e.g.

```
@Component
@Path("/hello")
```



```
    @GET
    public String message() {
```

```
        return "Hello";
    }

}
```

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

By default Jersey will be set up as a Servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. 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 servlet has an `@Order` which you can set with `spring.jersey.filter.order`. Both the Servlet and the Filter registrations can be given init parameters using `spring.jersey.init.*` to specify a map of properties.

There is a Jersey sample so you can see how to set things up. There is also a Jersey 1.x sample. Note that in the Jersey 1.x sample that the spring-boot maven plugin has been configured to unpack some Jersey jars so they can be scanned by the JAX-RS implementation (because the sample asks for them to be scanned in its `Filter` registration). You may need to do the same if any of your JAX-RS resources are packaged as nested jars.

27.3 Embedded servlet container support

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. Most developers will simply use the appropriate 'Starter POM' to obtain a fully configured instance. By default the embedded server will listen for HTTP requests on port `8080`.

27.3.1 Servlets, Filters, and listeners

When using an embedded servlet container you can register Servlets, Filters and all the listeners from the Servlet spec (e.g. `HttpSessionListener`) either by using Spring beans or by scanning for Servlet components.

Any `Servlet`, `Filter` or Servlet `*Listener` instance that is a Spring bean will be 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 will be mapped to `/`. In the case of multiple Servlet beans the bean name will be used as a path prefix. Filters will map to `/*`.

If convention-based mapping is not flexible enough you can use the `ServletRegistrationBean`, `FilterRegistrationBean` and `ServletListenerRegistrationBean` classes for complete control.

27.3.2 Servlet Context Initialization

Embedded servlet containers will not directly execute the Servlet 3.0+ `javax.servlet.ServletContainerInitializer` interface, or Spring's `org.springframework.web.WebApplicationInitializer` interface. This is an intentional design decision intended to reduce the risk that 3rd party libraries designed to run inside a war will 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 `org.springframework.boot.context.embedded.ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext`, and can easily be used as an adapter to an existing `WebApplicationInitializer` if necessary.

Scanning for Servlets, Filters, and listeners

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

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

27.3.3 The EmbeddedWebApplicationContext

Under the hood Spring Boot uses a new type of `ApplicationContext` for embedded servlet

`WebApplicationContext` that bootstraps itself by searching for a single `EmbeddedServletContainerFactory` bean. Usually a

`TomcatEmbeddedServletContainerFactory`, `JettyEmbeddedServletContainerFactory`, or `UndertowEmbeddedServletContainerFactory` will have been auto-configured.

You usually won't need to be aware of these implementation classes. Most applications will be auto-configured and the appropriate `ApplicationContext` and `EmbeddedServletContainerFactory` will be created on your behalf.

27.3.4 Customizing embedded servlet containers

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

Common server settings include:

- Network settings: listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, etc.
- Session settings: whether the session is persistent (`server.session.persistence`), session timeout (`server.session.timeout`), location of session data (`server.session.store-dir`) and session-cookie configuration (`server.session.cookie.*`).
- Error management: location of the error page (`server.error.path`), etc.
- 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.

See the `ServerProperties` class for a complete list.

Programmatic customization

If you need to configure your embedded servlet container programmatically you can register a

`EmbeddedServletContainerCustomizer` provides access to the `ConfigurableEmbeddedServletContainer` which includes numerous customization setter

methods.

```
import org.springframework.boot.context.embedded.*;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements EmbeddedServletContainerCustomizer {

    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.setPort(9000);
    }

}
```

Customizing ConfigurableEmbeddedServletContainer directly

If the above customization techniques are too limited, you can register the `TomcatEmbeddedServletContainerFactory`, `JettyEmbeddedServletContainerFactory` or `UndertowEmbeddedServletContainerFactory` bean yourself.

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory factory = new TomcatEmbeddedServletCont
    factory.setPort(9000);
    factory.setSessionTimeout(10, TimeUnit.MINUTES);
    factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND, "/notfound.html"));
    return factory;
}
```

Setters are provided for many configuration options. Several protected method 'hooks' are also provided should you need to do something more exotic. See the source code documentation for details.

27.3.5 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 Tomcat it should work if you use war packaging (i.e. an executable war will also be deployable to a standard container (not limited to, but including Tomcat). An executable jar will not work because of a hard coded file pattern in Tomcat.

- Jetty does not currently work as an embedded container with JSPs.
- Undertow does not support JSPs.

There is a JSP sample so you can see how to set things up.

Prev | Up | Next
26. Logging | Home | 28. Security