Java

```
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.http.HttpServletRequest;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

@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;
    }

}
```

Kotlin

```
import jakarta.servlet.RequestDispatcher
import jakarta.servlet.http.HttpServletRequest
import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.ControllerAdvice
import org.springframework.web.bind.annotation.ExceptionHandler
import org.springframework.web.bind.annotation.ResponseBody
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler

@ControllerAdvice(basePackageClasses = [SomeController::class])
class MyControllerAdvice : ResponseEntityExceptionHandler() {

    @ResponseBody
    @ExceptionHandler(MyException::class)
    fun handleControllerException(request: HttpServletRequest, ex: Throwable):
ResponseEntity<*> {
        val status = getStatus(request)
        return ResponseEntity(MyErrorBody(status.value(), ex.message), status)
    }

    private fun getStatus(request: HttpServletRequest): HttpStatus {
        val code = request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE) as Int
        val status = HttpStatus.resolve(code)
        return 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
             +- <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

```
import java.util.Map;

import jakarta.servlet.http.HttpServletRequest;

import org.springframework.boot.autoconfigure.web.servlet.error.ErrorViewResolver;
import org.springframework.http.HttpStatus;
import org.springframework.web.servlet.ModelAndView;

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;
    }

}
```

Kotlin

```
import jakarta.servlet.http.HttpServletRequest
import org.springframework.boot.autoconfigure.web.servlet.error.ErrorViewResolver
import org.springframework.http.HttpStatus
import org.springframework.web.servlet.ModelAndView

class MyErrorViewResolver : ErrorViewResolver {

    override fun resolveErrorView(request: HttpServletRequest, status: HttpStatus,
            model: Map<String, Any>): ModelAndView? {
        // Use the request or status to optionally return a ModelAndView
        if (status == HttpStatus.INSUFFICIENT_STORAGE) {
            // We could add custom model values here
            return 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 `ErrorPages`. This abstraction works directly with the underlying embedded servlet container and works even if you do not have a Spring MVC `DispatcherServlet`.

Java

```
import org.springframework.boot.web.server.ErrorPage;
import org.springframework.boot.web.server.ErrorPageRegistrar;
import org.springframework.boot.web.server.ErrorPageRegistry;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpStatus;

@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"));
    }

}
```

Kotlin

```
import org.springframework.boot.web.server.ErrorPage
import org.springframework.boot.web.server.ErrorPageRegistrar
import org.springframework.boot.web.server.ErrorPageRegistry
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.HttpStatus

@Configuration(proxyBeanMethods = false)
class MyErrorPagesConfiguration {

    @Bean
    fun errorPageRegistrar(): ErrorPageRegistrar {
        return ErrorPageRegistrar { registry: ErrorPageRegistry ->
registerErrorPages(registry) }
    }

    private fun registerErrorPages(registry: ErrorPageRegistry) {
        registry.addErrorPages(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

```
import java.util.EnumSet;

import jakarta.servlet.DispatcherType;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@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;
    }

}
```

Kotlin

```
import jakarta.servlet.DispatcherType
import org.springframework.boot.web.servlet.FilterRegistrationBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import java.util.EnumSet

@Configuration(proxyBeanMethods = false)
class MyFilterConfiguration {

    @Bean
    fun myFilter(): FilterRegistrationBean<MyFilter> {
        val registration = FilterRegistrationBean(MyFilter())
        // ...
        registration.setDispatcherTypes(EnumSet.allOf(DispatcherType::class.java))
        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

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration(proxyBeanMethods = false)
public class MyCorsConfiguration {

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

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

        };
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.CorsRegistry
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration(proxyBeanMethods = false)
class MyCorsConfiguration {

    @Bean
    fun corsConfigurer(): WebMvcConfigurer {
        return object : WebMvcConfigurer {
            override fun addCorsMappings(registry: CorsRegistry) {
                registry.addMapping("/api/**")
            }
        }
    }

}
```

8.1.2. 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 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:

```
import org.glassfish.jersey.server.ResourceConfig;

import org.springframework.stereotype.Component;

@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 `@Components` with HTTP resource annotations (`@GET` and others), as shown in the following example:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

import org.springframework.stereotype.Component;

@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.

8.1.3. 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.

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. 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 `jakarta.servlet.ServletContainerInitializer` interface or Spring's `org.springframework.web.WebApplicationInitializer` interface. 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 `org.springframework.boot.web.servlet.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:

```
import jakarta.servlet.ServletContext;

import org.springframework.boot.context.event.ApplicationStartedEvent;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationListener;
import org.springframework.web.context.WebApplicationContext;

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

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

Yaml

```
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

```
import org.springframework.boot.web.servlet.server.CookieSameSiteSupplier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MySameSiteConfiguration {

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

}
```

Kotlin

```
import org.springframework.boot.web.servlet.server.CookieSameSiteSupplier
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MySameSiteConfiguration {

    @Bean
    fun applicationCookieSameSiteSupplier(): CookieSameSiteSupplier {
        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 `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

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

Yaml

```
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

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import
org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class MyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

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

}
```

Kotlin

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory
import org.springframework.stereotype.Component

@Component
class MyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    override fun customize(server: ConfigurableServletWebServerFactory) {
        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

```
import java.time.Duration;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyTomcatWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

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

}
```

Kotlin

```
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.stereotype.Component
import java.time.Duration

@Component
class MyTomcatWebServerFactoryCustomizer :
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    override fun customize(server: TomcatServletWebServerFactory) {
        server.addConnectorCustomizers({ connector -> connector.asyncTimeout =
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 source code 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.

8.2. Reactive Web Applications

Spring Boot simplifies development of reactive web applications by providing auto-configuration for Spring Webflux.

8.2.1. The "Spring WebFlux Framework"

Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the servlet API, is fully asynchronous and non-blocking, and implements the Reactive Streams specification through the Reactor project.

Spring WebFlux comes in two flavors: functional and annotation-based. The annotation-based one is quite close to the Spring MVC model, as shown in the following example:

Java

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@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 Mono<User> getUser(@PathVariable Long userId) {
        return this.userRepository.findById(userId);
    }

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

    @DeleteMapping("/{userId}")
    public Mono<Void> deleteUser(@PathVariable Long userId) {
        return this.userRepository.deleteById(userId);
    }

}
```

Kotlin

```
import org.springframework.web.bind.annotation.DeleteMapping
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import reactor.core.publisher.Flux
import reactor.core.publisher.Mono

@RestController
@RequestMapping("/users")
class MyRestController(private val userRepository: UserRepository, private val
customerRepository: CustomerRepository) {

    @GetMapping("/{userId}")
    fun getUser(@PathVariable userId: Long): Mono<User?> {
        return userRepository.findById(userId)
    }

    @GetMapping("/{userId}/customers")
    fun getUserCustomers(@PathVariable userId: Long): Flux<Customer> {
        return userRepository.findById(userId).flatMapMany { user: User? ->
            customerRepository.findByUser(user)
        }
    }

    @DeleteMapping("/{userId}")
    fun deleteUser(@PathVariable userId: Long): Mono<Void> {
        return userRepository.deleteById(userId)
    }

}
```

WebFlux is part of the Spring Framework and detailed information is available in its reference documentation.

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

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.reactive.function.server.RequestPredicate;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static
org.springframework.web.reactive.function.server.RequestPredicates.accept;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;

@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

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

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

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.MediaType
import org.springframework.web.reactive.function.server.RequestPredicates.DELETE
import org.springframework.web.reactive.function.server.RequestPredicates.GET
import org.springframework.web.reactive.function.server.RequestPredicates.accept
import org.springframework.web.reactive.function.server.RouterFunction
import org.springframework.web.reactive.function.server.RouterFunctions
import org.springframework.web.reactive.function.server.ServerResponse

@Configuration(proxyBeanMethods = false)
class MyRoutingConfiguration {

    @Bean
    fun monoRouterFunction(userHandler: MyUserHandler): RouterFunction<ServerResponse>
{
        return RouterFunctions.route(
            GET("/{user}").and(ACCEPT_JSON), userHandler::getUser).andRoute(
            GET("/{user}/customers").and(ACCEPT_JSON),
userHandler::getUserCustomers).andRoute(
            DELETE("/{user}").and(ACCEPT_JSON), userHandler::deleteUser)
    }

    companion object {
        private val ACCEPT_JSON = accept(MediaType.APPLICATION_JSON)
    }

}
```

Java

```
import reactor.core.publisher.Mono;

import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;

@Component
public class MyUserHandler {

    public Mono<ServerResponse> getUser(ServerRequest request) {
        ...
    }

    public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
        ...
    }

    public Mono<ServerResponse> deleteUser(ServerRequest request) {
        ...
    }

}
```

Kotlin

```
import org.springframework.stereotype.Component
import org.springframework.web.reactive.function.server.ServerRequest
import org.springframework.web.reactive.function.server.ServerResponse
import reactor.core.publisher.Mono

@Component
class MyUserHandler {

    fun getUser(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

    fun getUserCustomers(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

    fun deleteUser(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

}
```

“WebFlux.fn” is part of the Spring Framework and detailed information is available in its reference

documentation.

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.

To get started, add the `spring-boot-starter-webflux` module to your application.

NOTE	Adding both `spring-boot-starter-web` and `spring-boot-starter-webflux` modules in your application results in Spring Boot auto-configuring Spring MVC, not WebFlux. This behavior has been chosen because many Spring developers add `spring-boot-starter-webflux` to their Spring MVC application to use the reactive `WebClient`. You can still enforce your choice by setting the chosen application type to `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`.

Spring WebFlux Auto-configuration

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

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

- Configuring codecs for `HttpMessageReader` and `HttpMessageWriter` instances (described later in this document).
- Support for serving static resources, including support for WebJars (described later in this document).

If you want to keep Spring Boot WebFlux features and you want to add additional WebFlux configuration, you can add your own `@Configuration` class of type `WebFluxConfigurer` but **without** `@EnableWebFlux`.

If you want to take complete control of Spring WebFlux, you can add your own `@Configuration` annotated with `@EnableWebFlux`.

Spring WebFlux Conversion Service

If you want to customize the `ConversionService` used by Spring WebFlux, you can provide a `WebFluxConfigurer` bean with an `addFormatters` method.

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

Property	`DateTimeFormatter`
`spring.webflux.format.date`	`ofLocalizedDate(FormatStyle.SHORT)`
`spring.webflux.format.time`	`ofLocalizedTime(FormatStyle.SHORT)`
`spring.webflux.format.date-time`	`ofLocalizedDateTime(FormatStyle.SHORT)`

HTTP Codecs with HttpMessageReaders and HttpMessageWriters

Spring WebFlux uses the `HttpMessageReader` and `HttpMessageWriter` interfaces to convert HTTP requests and responses. They are configured with `CodecConfigurer` to have sensible defaults by looking at the libraries available in your classpath.

Spring Boot provides dedicated configuration properties for codecs, `spring.codec.*`. It also applies further customization by using `CodecCustomizer` instances. For example, `spring.jackson.*` configuration keys are applied to the Jackson codec.

If you need to add or customize codecs, you can create a custom `CodecCustomizer` component, as shown in the following example:

Java

```
import org.springframework.boot.web.codec.CodecCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.codec.ServerSentEventHttpMessageReader;

@Configuration(proxyBeanMethods = false)
public class MyCodecsConfiguration {

    @Bean
    public CodecCustomizer myCodecCustomizer() {
        return (configurer) -> {
            configurer.registerDefaults(false);
            configurer.customCodecs().register(new
ServerSentEventHttpMessageReader());
            // ...
        };
    }

}
```

Kotlin

```
import org.springframework.boot.web.codec.CodecCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.http.codec.CodecConfigurer
import org.springframework.http.codec.ServerSentEventHttpMessageReader

class MyCodecsConfiguration {

    @Bean
    fun myCodecCustomizer(): CodecCustomizer {
        return CodecCustomizer { configurer: CodecConfigurer ->
            configurer.registerDefaults(false)
            configurer.customCodecs().register(ServerSentEventHttpMessageReader())
        }
    }

}
```

You can also leverage Boot's custom JSON serializers and deserializers.

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. It uses the `ResourceWebHandler` from Spring WebFlux so that you can modify that behavior by adding your own `WebFluxConfigurer` and overriding the `addResourceHandlers` method.

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

Properties

```
spring.webflux.static-path-pattern=/resources/**
```

Yaml

```
spring:
  webflux:
    static-path-pattern: "/resources/**"
```

You can also customize the static resource locations by using `spring.web.resources.static-locations`. Doing so replaces the default values with a list of directory locations. If you do so, the default welcome page detection switches to your custom locations. So, if there is an `index.html` in any of your locations on startup, it is the home page of the application.

In addition to the "standard" static resource locations listed 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.webflux.webjars-path-pattern` property.

TIP	Spring WebFlux applications do not strictly depend on the servlet API, so they cannot be deployed as war files and do not use the `src/main/webapp` directory.

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
`RouterFunctionMapping` for the Welcome Page	The welcome page support

Template Engines

As well as REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Spring WebFlux supports a variety of templating technologies, including Thymeleaf, FreeMarker, and Mustache.

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

- FreeMarker
- Thymeleaf
- Mustache

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

Error Handling

Spring Boot provides a `WebExceptionHandler` that handles all errors in a sensible way. Its position in the processing order is immediately before the handlers provided by WebFlux, which are considered last. 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 handler that renders the same data in HTML format. You can also provide your own HTML templates to display errors (see the next section).

Before customizing error handling in Spring Boot directly, you can leverage the RFC 7807 Problem Details support in Spring WebFlux. Spring WebFlux can produce custom error messages with the `application/problem+json` media type, like:

```
{
  "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.webflux.problemdetails.enabled` to `true`.

The first step to customizing this feature often involves using the existing mechanism but replacing or augmenting the error contents. For that, you can add a bean of type `ErrorAttributes`.

To change the error handling behavior, you can implement `ErrorWebExceptionHandler` and register a bean definition of that type. Because an `ErrorWebExceptionHandler` is quite low-level, Spring Boot also provides a convenient `AbstractErrorWebExceptionHandler` to let you handle errors in a WebFlux functional way, as shown in the following example:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.boot.autoconfigure.web.WebProperties;
import
org.springframework.boot.autoconfigure.web.reactive.error.AbstractErrorWebExceptionHan
dler;
import org.springframework.boot.web.reactive.error.ErrorAttributes;
import org.springframework.context.ApplicationContext;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.codec.ServerCodecConfigurer;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import org.springframework.web.reactive.function.server.ServerResponse.BodyBuilder;

@Component
public class MyErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler {

    public MyErrorWebExceptionHandler(ErrorAttributes errorAttributes, WebProperties
webProperties,
            ApplicationContext applicationContext, ServerCodecConfigurer
serverCodecConfigurer) {
        super(errorAttributes, webProperties.getResources(), applicationContext);
        setMessageReaders(serverCodecConfigurer.getReaders());
        setMessageWriters(serverCodecConfigurer.getWriters());
    }

    @Override
    protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes
errorAttributes) {
        return RouterFunctions.route(this::acceptsXml, this::handleErrorAsXml);
    }

    private boolean acceptsXml(ServerRequest request) {
        return request.headers().accept().contains(MediaType.APPLICATION_XML);
    }

    public Mono<ServerResponse> handleErrorAsXml(ServerRequest request) {
        BodyBuilder builder = ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR);
        // ... additional builder calls
        return builder.build();
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.web.WebProperties
import
org.springframework.boot.autoconfigure.web.reactive.error.AbstractErrorWebExceptionHan
dler
import org.springframework.boot.web.reactive.error.ErrorAttributes
import org.springframework.context.ApplicationContext
import org.springframework.http.HttpStatus
import org.springframework.http.MediaType
import org.springframework.http.codec.ServerCodecConfigurer
import org.springframework.stereotype.Component
import org.springframework.web.reactive.function.server.RouterFunction
import org.springframework.web.reactive.function.server.RouterFunctions
import org.springframework.web.reactive.function.server.ServerRequest
import org.springframework.web.reactive.function.server.ServerResponse
import reactor.core.publisher.Mono

@Component
class MyErrorWebExceptionHandler(
        errorAttributes: ErrorAttributes, webProperties: WebProperties,
        applicationContext: ApplicationContext, serverCodecConfigurer:
ServerCodecConfigurer
) : AbstractErrorWebExceptionHandler(errorAttributes, webProperties.resources,
applicationContext) {

    init {
        setMessageReaders(serverCodecConfigurer.readers)
        setMessageWriters(serverCodecConfigurer.writers)
    }

    override fun getRoutingFunction(errorAttributes: ErrorAttributes):
RouterFunction<ServerResponse> {
        return RouterFunctions.route(this::acceptsXml, this::handleErrorAsXml)
    }

    private fun acceptsXml(request: ServerRequest): Boolean {
        return request.headers().accept().contains(MediaType.APPLICATION_XML)
    }

    fun handleErrorAsXml(request: ServerRequest): Mono<ServerResponse> {
        val builder = ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR)
        // ... additional builder calls
        return builder.build()
    }

}
```

For a more complete picture, you can also subclass `DefaultErrorWebExceptionHandler` directly and override specific methods.

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 views that resolve from `error/*`, for example by adding files to a `/error` directory. Error pages can either be static HTML (that is, added under any of the static resource directories) or built with templates. The name of the file should be the exact status code, a status code series mask, or `error` for a default if nothing else matches. Note that the path to the default error view is `error/error`, whereas with Spring MVC the default error view is `error`.

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
             +- <other public assets>
```

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

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

Web Filters

Spring WebFlux provides a `WebFilter` interface that can be implemented to filter HTTP request-response exchanges. `WebFilter` beans found in the application context will be automatically used to filter each exchange.

Where the order of the filters is important they can implement `Ordered` or be annotated with `@Order`. Spring Boot auto-configuration may configure web filters for you. When it does so, the orders shown in the following table will be used:

Web Filter	Order
`WebFilterChainProxy` (Spring Security)	`-100`
`HttpExchangesWebFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

8.2.2. Embedded Reactive Server Support

Spring Boot includes support for the following embedded reactive web servers: Reactor Netty, Tomcat, Jetty, and Undertow. Most developers use the appropriate "Starter" to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port 8080.

Customizing Reactive Servers

Common reactive web server 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.
- 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 such as `server.netty.*` offer server-specific customizations.

> **TIP** See the `ServerProperties` class for a complete list.

Programmatic Customization

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

Java

```
import
org.springframework.boot.web.reactive.server.ConfigurableReactiveWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<ConfigurableReactiveWebServerFactory> {

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

}
```

Kotlin

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import
org.springframework.boot.web.reactive.server.ConfigurableReactiveWebServerFactory
import org.springframework.stereotype.Component

@Component
class MyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<ConfigurableReactiveWebServerFactory> {

    override fun customize(server: ConfigurableReactiveWebServerFactory) {
        server.setPort(9000)
    }

}
```

`JettyReactiveWebServerFactory`, `NettyReactiveWebServerFactory`, `TomcatReactiveWebServerFactory`, and `UndertowReactiveWebServerFactory` are dedicated variants of `ConfigurableReactiveWebServerFactory` that have additional customization setter methods for Jetty, Reactor Netty, Tomcat, and Undertow respectively. The following example shows how to customize `NettyReactiveWebServerFactory` that provides access to Reactor Netty-specific configuration options:

Java

```
import java.time.Duration;

import org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyNettyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<NettyReactiveWebServerFactory> {

    @Override
    public void customize(NettyReactiveWebServerFactory factory) {
        factory.addServerCustomizers((server) ->
server.idleTimeout(Duration.ofSeconds(20)));
    }

}
```

Kotlin

```
import org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.stereotype.Component
import java.time.Duration

@Component
class MyNettyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<NettyReactiveWebServerFactory> {

    override fun customize(factory: NettyReactiveWebServerFactory) {
        factory.addServerCustomizers({ server ->
server.idleTimeout(Duration.ofSeconds(20)) })
    }

}
```

Customizing ConfigurableReactiveWebServerFactory Directly

For more advanced use cases that require you to extend from `ReactiveWebServerFactory`, 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 source code documentation for details.

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

8.2.3. Reactive Server Resources Configuration

When auto-configuring a Reactor Netty or Jetty server, Spring Boot will create specific beans that will provide HTTP resources to the server instance: `ReactorResourceFactory` or `JettyResourceFactory`.

By default, those resources will be also shared with the Reactor Netty and Jetty clients for optimal performances, given:

- the same technology is used for server and client
- the client instance is built using the `WebClient.Builder` bean auto-configured by Spring Boot

Developers can override the resource configuration for Jetty and Reactor Netty by providing a custom `ReactorResourceFactory` or `JettyResourceFactory` bean - this will be applied to both clients and servers.

You can learn more about the resource configuration on the client side in the WebClient Runtime section.

8.3. Graceful Shutdown

Graceful shutdown is supported with all four embedded web servers (Jetty, Reactor Netty, Tomcat, and Undertow) and with both reactive and servlet-based web applications. It occurs as part of closing the application context and is performed in the earliest phase of stopping `SmartLifecycle` beans. This stop processing uses a timeout which provides a grace period during which existing requests will be allowed to complete but no new requests will be permitted.

The exact way in which new requests are not permitted varies depending on the web server that is being used. Implementations may stop accepting requests at the network layer, or they may return a response with a specific HTTP status code or HTTP header. The use of persistent connections can also change the way that requests stop being accepted.

TIP	To learn about more the specific method used with your web server, see the `shutDownGracefully` javadoc for TomcatWebServer, NettyWebServer, JettyWebServer or UndertowWebServer.

Jetty, Reactor Netty, and Tomcat will stop accepting new requests at the network layer. Undertow will accept new connections but respond immediately with a service unavailable (503) response.

NOTE	Graceful shutdown with Tomcat requires Tomcat 9.0.33 or later.

To enable graceful shutdown, configure the `server.shutdown` property, as shown in the following example:

Properties

```
server.shutdown=graceful
```

Yaml

```
server:
  shutdown: "graceful"
```

To configure the timeout period, configure the `spring.lifecycle.timeout-per-shutdown-phase` property, as shown in the following example:

Properties

```
spring.lifecycle.timeout-per-shutdown-phase=20s
```

Yaml

```
spring:
  lifecycle:
    timeout-per-shutdown-phase: "20s"
```

IMPORTANT	Using graceful shutdown with your IDE may not work properly if it does not send a proper `SIGTERM` signal. See the documentation of your IDE for more details.

8.4. Spring Security

If Spring Security is on the classpath, then web applications are secured by default. Spring Boot relies on Spring Security's content-negotiation strategy to determine whether to use `httpBasic` or `formLogin`. To add method-level security to a web application, you can also add `@EnableGlobalMethodSecurity` with your desired settings. Additional information can be found in the Spring Security Reference Guide.

The default `UserDetailsService` has a single user. The user name is `user`, and the password is random and is printed at WARN level when the application starts, as shown in the following example:

```
Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35

This generated password is for development use only. Your security configuration must
be updated before running your application in production.
```

NOTE	If you fine-tune your logging configuration, ensure that the `org.springframework.boot.autoconfigure.security` category is set to log `WARN`-level messages. Otherwise, the default password is not printed.

You can change the username and password by providing a `spring.security.user.name` and `spring.security.user.password`.

The basic features you get by default in a web application are:

- A `UserDetailsService` (or `ReactiveUserDetailsService` in case of a WebFlux application) bean with in-memory store and a single user with a generated password (see `SecurityProperties.User` for the properties of the user).
- Form-based login or HTTP Basic security (depending on the `Accept` header in the request) for the entire application (including actuator endpoints if actuator is on the classpath).
- A `DefaultAuthenticationEventPublisher` for publishing authentication events.

You can provide a different `AuthenticationEventPublisher` by adding a bean for it.

8.4.1. MVC Security

The default security configuration is implemented in `SecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `SecurityAutoConfiguration` imports `SpringBootWebSecurityConfiguration` for web security and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications.

To switch off the default web application security configuration completely or to combine multiple Spring Security components such as OAuth2 Client and Resource Server, add a bean of type `SecurityFilterChain` (doing so does not disable the `UserDetailsService` configuration or Actuator's security). To also switch off the `UserDetailsService` configuration, you can add a bean of type `UserDetailsService`, `AuthenticationProvider`, or `AuthenticationManager`.

The auto-configuration of a `UserDetailsService` will also back off any of the following Spring Security modules is on the classpath:

- `spring-security-oauth2-client`
- `spring-security-oauth2-resource-server`
- `spring-security-saml2-service-provider`

To use `UserDetailsService` in addition to one or more of these dependencies, define your own `InMemoryUserDetailsManager` bean.

Access rules can be overridden by adding a custom `SecurityFilterChain` bean. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `RequestMatcher` that is based on the `management.endpoints.web.base-path` property. `PathRequest` can be used to create a `RequestMatcher` for resources in commonly used locations.

8.4.2. WebFlux Security

Similar to Spring MVC applications, you can secure your WebFlux applications by adding the `spring-boot-starter-security` dependency. The default security configuration is implemented in `ReactiveSecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `ReactiveSecurityAutoConfiguration` imports `WebFluxSecurityConfiguration` for web security and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications.

To switch off the default web application security configuration completely, you can add a bean of type `WebFilterChainProxy` (doing so does not disable the `UserDetailsService` configuration or Actuator's security). To also switch off the `UserDetailsService` configuration, you can add a bean of type `ReactiveUserDetailsService` or `ReactiveAuthenticationManager`.

The auto-configuration will also back off when any of the following Spring Security modules is on the classpath:

- `spring-security-oauth2-client`
- `spring-security-oauth2-resource-server`

To use `ReactiveUserDetailsService` in addition to one or more of these dependencies, define your own `MapReactiveUserDetailsService` bean.

Access rules and the use of multiple Spring Security components such as OAuth 2 Client and Resource Server can be configured by adding a custom `SecurityWebFilterChain` bean. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `ServerWebExchangeMatcher` that is based on the `management.endpoints.web.base-path` property.

`PathRequest` can be used to create a `ServerWebExchangeMatcher` for resources in commonly used locations.

For example, you can customize your security configuration by adding something like:

Java

```
import org.springframework.boot.autoconfigure.security.reactive.PathRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration(proxyBeanMethods = false)
public class MyWebFluxSecurityConfiguration {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http.authorizeExchange((exchange) -> {

exchange.matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll();
            exchange.pathMatchers("/foo", "/bar").authenticated();
        });
        http.formLogin(withDefaults());
        return http.build();
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.security.reactive.PathRequest
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.Customizer.withDefaults
import org.springframework.security.config.web.server.ServerHttpSecurity
import org.springframework.security.web.server.SecurityWebFilterChain

@Configuration(proxyBeanMethods = false)
class MyWebFluxSecurityConfiguration {

    @Bean
    fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
        http.authorizeExchange { spec ->

spec.matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
            spec.pathMatchers("/foo", "/bar").authenticated()
        }
        http.formLogin(withDefaults())
        return http.build()
    }

}
```

8.4.3. OAuth2

OAuth2 is a widely used authorization framework that is supported by Spring.

Client

If you have `spring-security-oauth2-client` on your classpath, you can take advantage of some auto-configuration to set up OAuth2/Open ID Connect clients. This configuration makes use of the properties under `OAuth2ClientProperties`. The same properties are applicable to both servlet and reactive applications.

You can register multiple OAuth2 clients and providers under the `spring.security.oauth2.client` prefix, as shown in the following example:

Properties

```
spring.security.oauth2.client.registration.my-login-client.client-id=abcd
spring.security.oauth2.client.registration.my-login-client.client-secret=password
spring.security.oauth2.client.registration.my-login-client.client-name=Client for
OpenID Connect
spring.security.oauth2.client.registration.my-login-client.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-login-
client.scope=openid,profile,email,phone,address
spring.security.oauth2.client.registration.my-login-client.redirect-
uri={baseUrl}/login/oauth2/code/{registrationId}
```

```
spring.security.oauth2.client.registration.my-login-client.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-login-client.authorization-grant-
type=authorization_code

spring.security.oauth2.client.registration.my-client-1.client-id=abcd
spring.security.oauth2.client.registration.my-client-1.client-secret=password
spring.security.oauth2.client.registration.my-client-1.client-name=Client for user
scope
spring.security.oauth2.client.registration.my-client-1.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-1.scope=user
spring.security.oauth2.client.registration.my-client-1.redirect-
uri={baseUrl}/authorized/user
spring.security.oauth2.client.registration.my-client-1.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-client-1.authorization-grant-
type=authorization_code

spring.security.oauth2.client.registration.my-client-2.client-id=abcd
spring.security.oauth2.client.registration.my-client-2.client-secret=password
spring.security.oauth2.client.registration.my-client-2.client-name=Client for email
scope
spring.security.oauth2.client.registration.my-client-2.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-2.scope=email
spring.security.oauth2.client.registration.my-client-2.redirect-
uri={baseUrl}/authorized/email
spring.security.oauth2.client.registration.my-client-2.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-client-2.authorization-grant-
type=authorization_code

spring.security.oauth2.client.provider.my-oauth-provider.authorization-uri=https://my-
auth-server.com/oauth2/authorize
spring.security.oauth2.client.provider.my-oauth-provider.token-uri=https://my-auth-
server.com/oauth2/token
spring.security.oauth2.client.provider.my-oauth-provider.user-info-uri=https://my-
auth-server.com/userinfo
spring.security.oauth2.client.provider.my-oauth-provider.user-info-authentication-
method=header
spring.security.oauth2.client.provider.my-oauth-provider.jwk-set-uri=https://my-auth-
server.com/oauth2/jwks
spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name
```

Yaml

```
spring:
  security:
    oauth2:
      client:
        registration:
          my-login-client:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for OpenID Connect"
            provider: "my-oauth-provider"
            scope: "openid,profile,email,phone,address"
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

          my-client-1:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for user scope"
            provider: "my-oauth-provider"
            scope: "user"
            redirect-uri: "{baseUrl}/authorized/user"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

          my-client-2:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for email scope"
            provider: "my-oauth-provider"
            scope: "email"
            redirect-uri: "{baseUrl}/authorized/email"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

        provider:
          my-oauth-provider:
            authorization-uri: "https://my-auth-server.com/oauth2/authorize"
            token-uri: "https://my-auth-server.com/oauth2/token"
            user-info-uri: "https://my-auth-server.com/userinfo"
            user-info-authentication-method: "header"
            jwk-set-uri: "https://my-auth-server.com/oauth2/jwks"
            user-name-attribute: "name"
```

For OpenID Connect providers that support OpenID Connect discovery, the configuration can be further simplified. The provider needs to be configured with an `issuer-uri` which is the URI that it asserts as its Issuer Identifier. For example, if the `issuer-uri` provided is "https://example.com", then an "OpenID Provider Configuration Request" will be made to "https://example.com/.well-

known/openid-configuration". The result is expected to be an "OpenID Provider Configuration Response". The following example shows how an OpenID Connect Provider can be configured with the `issuer-uri`:

Properties

```
spring.security.oauth2.client.provider.oidc-provider.issuer-uri=https://dev-
123456.oktapreview.com/oauth2/default/
```

Yaml

```
spring:
  security:
    oauth2:
      client:
        provider:
          oidc-provider:
            issuer-uri: "https://dev-123456.oktapreview.com/oauth2/default/"
```

By default, Spring Security's `OAuth2LoginAuthenticationFilter` only processes URLs matching `/login/oauth2/code/*`. If you want to customize the `redirect-uri` to use a different pattern, you need to provide configuration to process that custom pattern. For example, for servlet applications, you can add your own `SecurityFilterChain` that resembles the following:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration(proxyBeanMethods = false)
@EnableWebSecurity
public class MyOAuthClientConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http
            .authorizeHttpRequests((requests) -> requests
                .anyRequest().authenticated()
            )
            .oauth2Login((login) -> login
                .redirectionEndpoint((endpoint) -> endpoint
                    .baseUri("/login/oauth2/callback/*")
                )
            );
        return http.build();
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
import org.springframework.security.config.annotation.web.invoke
import org.springframework.security.web.SecurityFilterChain

@Configuration(proxyBeanMethods = false)
@EnableWebSecurity
open class MyOAuthClientConfiguration {

    @Bean
    open fun securityFilterChain(http: HttpSecurity): SecurityFilterChain {
        http {
            authorizeHttpRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2Login {
                redirectionEndpoint {
                    baseUri = "/login/oauth2/callback/*"
                }
            }
        }
        return http.build()
    }

}
```

TIP

Spring Boot auto-configures an `InMemoryOAuth2AuthorizedClientService` which is used by Spring Security for the management of client registrations. The `InMemoryOAuth2AuthorizedClientService` has limited capabilities and we recommend using it only for development environments. For production environments, consider using a `JdbcOAuth2AuthorizedClientService` or creating your own implementation of `OAuth2AuthorizedClientService`.

OAuth2 Client Registration for Common Providers

For common OAuth2 and OpenID providers, including Google, Github, Facebook, and Okta, we provide a set of provider defaults (`google`, `github`, `facebook`, and `okta`, respectively).

If you do not need to customize these providers, you can set the `provider` attribute to the one for which you need to infer defaults. Also, if the key for the client registration matches a default supported provider, Spring Boot infers that as well.

In other words, the two configurations in the following example use the Google provider:

Properties

```
spring.security.oauth2.client.registration.my-client.client-id=abcd
spring.security.oauth2.client.registration.my-client.client-secret=password
spring.security.oauth2.client.registration.my-client.provider=google
spring.security.oauth2.client.registration.google.client-id=abcd
spring.security.oauth2.client.registration.google.client-secret=password
```

Yaml

```
spring:
  security:
    oauth2:
      client:
        registration:
          my-client:
            client-id: "abcd"
            client-secret: "password"
            provider: "google"
          google:
            client-id: "abcd"
            client-secret: "password"
```

Resource Server

If you have `spring-security-oauth2-resource-server` on your classpath, Spring Boot can set up an OAuth2 Resource Server. For JWT configuration, a JWK Set URI or OIDC Issuer URI needs to be specified, as shown in the following examples:

Properties

```
spring.security.oauth2.resourceserver.jwt.jwk-set-
uri=https://example.com/oauth2/default/v1/keys
```

Yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: "https://example.com/oauth2/default/v1/keys"
```

Properties

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=https://dev-
123456.oktapreview.com/oauth2/default/
```

Yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: "https://dev-123456.oktapreview.com/oauth2/default/"
```

NOTE If the authorization server does not support a JWK Set URI, you can configure the resource server with the Public Key used for verifying the signature of the JWT. This can be done using the `spring.security.oauth2.resourceserver.jwt.public-key-location` property, where the value needs to point to a file containing the public key in the PEM-encoded x509 format.

The `spring.security.oauth2.resourceserver.jwt.audiences` property can be used to specify the expected values of the aud claim in JWTs. For example, to require JWTs to contain an aud claim with the value `my-audience`:

Properties

```
spring.security.oauth2.resourceserver.jwt.audiences[0]=my-audience
```

Yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          audiences:
            - "my-audience"
```

The same properties are applicable for both servlet and reactive applications. Alternatively, you can define your own `JwtDecoder` bean for servlet applications or a `ReactiveJwtDecoder` for reactive applications.

In cases where opaque tokens are used instead of JWTs, you can configure the following properties to validate tokens through introspection:

Properties

```
spring.security.oauth2.resourceserver.opaquetoken.introspection-
uri=https://example.com/check-token
spring.security.oauth2.resourceserver.opaquetoken.client-id=my-client-id
spring.security.oauth2.resourceserver.opaquetoken.client-secret=my-client-secret
```

Yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        opaquetoken:
          introspection-uri: "https://example.com/check-token"
          client-id: "my-client-id"
          client-secret: "my-client-secret"
```

Again, the same properties are applicable for both servlet and reactive applications. Alternatively, you can define your own `OpaqueTokenIntrospector` bean for servlet applications or a `ReactiveOpaqueTokenIntrospector` for reactive applications.

Authorization Server

If you have `spring-security-oauth2-authorization-server` on your classpath, you can take advantage of some auto-configuration to set up a Servlet-based OAuth2 Authorization Server.

You can register multiple OAuth2 clients under the `spring.security.oauth2.authorizationserver.client` prefix, as shown in the following example:

Properties

```
spring.security.oauth2.authorizationserver.client.my-client-1.registration.client-
id=abcd
spring.security.oauth2.authorizationserver.client.my-client-1.registration.client-
secret={noop}secret1
spring.security.oauth2.authorizationserver.client.my-client-1.registration.client-
authentication-methods[0]=client_secret_basic
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.authorization-grant-types[0]=authorization_code
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.authorization-grant-types[1]=refresh_token
spring.security.oauth2.authorizationserver.client.my-client-1.registration.redirect-
uris[0]=https://my-client-1.com/login/oauth2/code/abcd
spring.security.oauth2.authorizationserver.client.my-client-1.registration.redirect-
uris[1]=https://my-client-1.com/authorized
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[0]=openid
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[1]=profile
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[2]=email
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[3]=phone
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[4]=address
spring.security.oauth2.authorizationserver.client.my-client-1.require-authorization-
consent=true
spring.security.oauth2.authorizationserver.client.my-client-2.registration.client-
id=efgh
spring.security.oauth2.authorizationserver.client.my-client-2.registration.client-
secret={noop}secret2
spring.security.oauth2.authorizationserver.client.my-client-2.registration.client-
authentication-methods[0]=client_secret_jwt
spring.security.oauth2.authorizationserver.client.my-client-
2.registration.authorization-grant-types[0]=client_credentials
spring.security.oauth2.authorizationserver.client.my-client-
2.registration.scopes[0]=user.read
spring.security.oauth2.authorizationserver.client.my-client-
2.registration.scopes[1]=user.write
spring.security.oauth2.authorizationserver.client.my-client-2.jwk-set-uri=https://my-
client-2.com/jwks
spring.security.oauth2.authorizationserver.client.my-client-2.token-endpoint-
authentication-signing-algorithm=RS256
```

Yaml

```
spring:
  security:
    oauth2:
      authorizationserver:
        client:
          my-client-1:
            registration:
              client-id: "abcd"
              client-secret: "{noop}secret1"
              client-authentication-methods:
                - "client_secret_basic"
              authorization-grant-types:
                - "authorization_code"
                - "refresh_token"
              redirect-uris:
                - "https://my-client-1.com/login/oauth2/code/abcd"
                - "https://my-client-1.com/authorized"
              scopes:
                - "openid"
                - "profile"
                - "email"
                - "phone"
                - "address"
            require-authorization-consent: true
          my-client-2:
            registration:
              client-id: "efgh"
              client-secret: "{noop}secret2"
              client-authentication-methods:
                - "client_secret_jwt"
              authorization-grant-types:
                - "client_credentials"
              scopes:
                - "user.read"
                - "user.write"
            jwk-set-uri: "https://my-client-2.com/jwks"
            token-endpoint-authentication-signing-algorithm: "RS256"
```

NOTE The `client-secret` property must be in a format that can be matched by the configured `PasswordEncoder`. The default instance of `PasswordEncoder` is created via `PasswordEncoderFactories.createDelegatingPasswordEncoder()`.

The auto-configuration Spring Boot provides for Spring Authorization Server is designed for getting started quickly. Most applications will require customization and will want to define several beans to override auto-configuration.

The following components can be defined as beans to override auto-configuration specific to Spring Authorization Server:

- `RegisteredClientRepository`
- `AuthorizationServerSettings`
- `SecurityFilterChain`
- `com.nimbusds.jose.jwk.source.JWKSource<com.nimbusds.jose.proc.SecurityContext>`
- `JwtDecoder`

TIP	Spring Boot auto-configures an `InMemoryRegisteredClientRepository` which is used by Spring Authorization Server for the management of registered clients. The `InMemoryRegisteredClientRepository` has limited capabilities and we recommend using it only for development environments. For production environments, consider using a `JdbcRegisteredClientRepository` or creating your own implementation of `RegisteredClientRepository`.

Additional information can be found in the Getting Started chapter of the Spring Authorization Server Reference Guide.

8.4.4. SAML 2.0

Relying Party

If you have `spring-security-saml2-service-provider` on your classpath, you can take advantage of some auto-configuration to set up a SAML 2.0 Relying Party. This configuration makes use of the properties under `Saml2RelyingPartyProperties`.

A relying party registration represents a paired configuration between an Identity Provider, IDP, and a Service Provider, SP. You can register multiple relying parties under the `spring.security.saml2.relyingparty` prefix, as shown in the following example: