Red Hat support for Spring Boot 2.4

Reactive Application Development Guide

Build reactive applications with Spring Boot Starters based on non-blocking networking components provided by Eclipse Vert.x that run on OpenShift and on stand-alone RHEL

Last Updated: 2022-03-11

Red Hat support for Spring Boot 2.4 Reactive Application Development Guide

Build reactive applications with Spring Boot Starters based on non-blocking networking components provided by Eclipse Vert.x that run on OpenShift and on stand-alone RHEL

Legal Notice

Abstract

This guide provides details about using Spring Boot with Eclipse Vert.x to develop cloud-native non-blocking reactive applications.

Table of Contents

PREFACE

Develop reactive applications with messaging, streaming and authentication capabilities using Spring Boot Starters with reactive Eclipse Vert.x networking components. You can deploy your applications to OpenShift and standalone RHEL.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. To provide feedback, you can highlight the text in a document and add comments.

This section explains how to submit feedback.

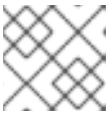
Prerequisites

- You are logged in to the Red Hat Customer Portal.
- In the Red Hat Customer Portal, view the document in **Multi-page HTML** format.

Procedure

To provide your feedback, perform the following steps:

1. Click the **Feedback** button in the top-right corner of the document to see existing feedback.



NOTE

The feedback feature is enabled only in the **Multi-page HTML** format.

2. Highlight the section of the document where you want to provide feedback.
3. Click the **Add Feedback** pop-up that appears near the highlighted text.
   A text box appears in the feedback section on the right side of the page.
4. Enter your feedback in the text box and click **Submit**.
   A documentation issue is created.
5. To view the issue, click the issue tracker link in the feedback view.

CHAPTER 1. CONFIGURING YOUR APPLICATION TO USE SPRING BOOT

Configure your application to use dependencies provided with Red Hat build of Spring Boot. By using the BOM to manage your dependencies, you ensure that your applications always uses the product version of these dependencies that Red Hat provides support for. Reference the Spring Boot BOM (Bill of Materials) artifact in the **pom.xml** file at the root directory of your application. You can use the BOM in your application project in 2 different ways:

- As a dependency in the **<dependencyManagement>** section of the **pom.xml**. When using the BOM as a dependency, your project inherits the version settings for all Spring Boot dependencies from the **<dependencyManagement>** section of the BOM.
- As a parent BOM in the **<parent>** section of the **pom.xml**. When using the BOM as a parent, the **pom.xml** of your project inherits the following configuration values from the parent BOM:
  - versions of all Spring Boot dependencies in the **<dependencyManagement>** section
  - versions plugins in the **<pluginManagement>** section
  - the URLs and names of repositories in the **<repositories>** section
  - the URLs and name of the repository that contains the Spring Boot plugin in the **<pluginRepositories>** section

1.1. PREREQUISITES

- A Maven-based application project that you configure using a **pom.xml** file.
- Access to the Red Hat JBoss Middleware General Availability Maven Repository .

1.2. USING THE SPRING BOOT BOM TO MANAGE DEPENDENCY VERSIONS

Manage versions of Spring Boot product dependencies in your application project using the product BOM.

Procedure

1. Add the **dev.snowdrop:snowdrop-dependencies** artifact to the **<dependencyManagement>** section of the **pom.xml** of your project, and specify the values of the **<type>** and **<scope>** attributes:

```
<project>
 ...
 <dependencyManagement>
  <dependencies>
   <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>snowdrop-dependencies</artifactId>
    <version>2.4.9.Final-redhat-00001</version>
    <type>pom</type>
    <scope>import</scope>
   </dependency>
```

```
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

2. Include the following properties to track the version of the Spring Boot Maven Plugin that you are using:

```
<project>
  ...
  <properties>
    <spring-boot-maven-plugin.version>2.4.9</spring-boot-maven-plugin.version>
  </properties>
  ...
</project>
```

3. Specify the names and URLs of repositories containing the BOM and the supported Spring Boot Starters and the Spring Boot Maven plugin:

```
  <!-- Specify the repositories containing Spring Boot artifacts. -->
  <repositories>
    <repository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </repository>
  </repositories>

  <!-- Specify the repositories containing the plugins used to execute the build of your application. -->
  <pluginRepositories>
    <pluginRepository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </pluginRepository>
  </pluginRepositories>
```

4. Add **spring-boot-maven-plugin** as the plugin that Maven uses to package your application.

```
<project>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>${spring-boot-maven-plugin.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
```

```
          </execution>
        </executions>
          <configuration>
            <redeploy>true</redeploy>
          </configuration>
      </plugin>
      ...
    </plugins>
  ...
 </build>
 ...
</project>
```

1.3. USING THE SPRING BOOT BOM TO AS A PARENT BOM OF YOUR APPLICATION

Automatically manage the:

- versions of product dependencies
- version of the Spring Boot Maven plugin
- configuration of Maven repositories containing the product artifacts and plugins

that you use in your application project by including the product Spring Boot BOM as a parent BOM of your project. This method provides an alternative to using the BOM as a dependency of your application.

Procedure

1. Add the **dev.snowdrop:snowdrop-dependencies** artifact to the **<parent>** section of the **pom.xml**:

```
<project>
  ...
  <parent>
    <groupId>dev.snowdrop</groupId>
    <artifactId>snowdrop-dependencies</artifactId>
    <version>2.4.9.Final-redhat-00001</version>
  </parent>
  ...
</project>
```

2. Add **spring-boot-maven-plugin** as the plugin that Maven uses to package your application to the **<build>** section of the **pom.xml**. The plugin version is automatically managed by the parent BOM.

```
<project>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-maven-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <redeploy>true</redeploy>
        </configuration>
      </plugin>
      ...
    </plugins>
  ...
 </build>
 ...
</project>
```

1.4. RELATED INFORMATION

- For more information about packaging your Spring Boot application, see the Spring Boot Maven Plugin documentation.

CHAPTER 2. DEVELOPING REACTIVE APPLICATIONS USING SPRING BOOT WITH ECLIPSE VERT.X

This section provides an introduction to developing applications in a reactive way using Spring Boot starters based on Spring Boot and Eclipse Vert.x. The following examples demonstrate how you can use the starters to create reactive applications.

2.1. INTRODUCTION TO SPRING BOOT WITH ECLIPSE VERT.X

The Spring reactive stack is build on Project Reactor, a reactive library that implements backpressure and is compliant with the Reactive Streams specification. It provides the **Flux** and **Mono** functional API types that enable asynchronous event stream processing.

On top of Project Reactor, Spring provides WebFlux, an asynchronous event-driven web application framework. While WebFlux is designed to work primarily with Reactor Netty, it can also operate with other reactive HTTP servers, such as Eclipse Vert.x.

Spring WebFlux and Reactor enable you to create applications that are:

- Non-blocking: The application continues to handle further requests when waiting for a response from a remote component or service that is required to complete the current request.
- Asynchronous: the application responds to events from an event stream by generating response events and publishing them back to the event stream where they can be picked up by other clients in the application.
- Event-driven: The application responds to events generated by the user or by another service, such as mouse clicks, HTTP requests, or new files being added to a storage.
- Scalable: Increasing the number of Publishers or Subscribers to match the required event processing capacity of an application only results in a slight increase in the complexity of routing requests between individual clients in the application. Reactive applications can handle large numbers of events using fewer computing and networking resources as compared to other application programming models.
- Resilient: The application can handle failure of services it depend on without a negative impact on its overall quality of service.

Additional advantages of using Spring WebFlux include:

Similarity with SpringMVC

The SpringMVC API types and WebFlux API types are similar, and it is easy for developers to apply knowledge of SpringMVC to programming applications with WebFlux.

The Spring Reactive offering by Red Hat brings the benefits of Reactor and WebFlux to OpenShift and stand-alone RHEL, and introduces a set of Eclipse Vert.x extensions for the WebFLux framework. This allows you to retain the level of abstraction and rapid prototyping capabilities of Spring Boot, and provides an asynchronous IO API that handles the network communications between the services in your application in a fully reactive manner.

Annotated controllers support

WebFlux retains the endpoint controller annotations introduced by SpringMVC (Both SpringMVC and WebFlux support reactive RxJava2 and Reactor return types).

Functional programming support

Reactor interacts with the Java 8 Functional API, as well as **CompletablebFuture**, and **Stream** APIs. In addition to annotation-based endpoints, WebFlux also supports functional endpoints.

Additional resources

See the following resources for additional in-depth information on the implementation details of technologies that are part of the Spring Reactive stack:

- The Reactive Manifesto
- Reactive Streams specification
- Spring Framework reference documentation: Web Applications on Reactive Stack
- Reactor Netty documentation
- API Reference page for the **Mono** class in Project Reactor Documentation
- API Reference page for the **Flux** class in Project Reactor Documentation

2.2. REACTIVE SPRING WEB

The **spring-web** module provides the foundational elements of the reactive capabilities of Spring WebFlux, including:

- HTTP abstractions provided by the **HttpHandler** API
- Reactive Streams adapters for supported servers (Eclipse Vert.x, Undertow and others)
- Codecs for encoding and decoding event stream data. This includes:
  - **DataBuffer**, an abstraction for different types of byte buffer representations (Netty **ByteBuf**, **java.nio.ByteBuffer**, as well as others)
  - Low-level contracts to encode and decode content independent of HTTP
  - **HttpMessageReader** and **HTTPMessageWriter** contracts to encode and decode HTTP message content
- The **WebHandler** API (a counterpart to the Servlet 3.1 I/O API that uses non-blocking contracts).

When designing your web application, you can choose between 2 programming models that Spring WebFlux provides:

Annotated Controllers

Annotated controllers in Spring WebFlux are consistent with Spring MVC, and are based on the same annotations from the **spring-web** module. In addition to the **spring-web** module from SpringMVC, its WebFlux counterpart also supports reactive **@RequestBody** arguments.

Functional Endpoints

Functional endpoints provided by spring WebFlux on Java 8 Lambda expressions and functional APIs, this programming model relies on a dedicated library (Reactor, in this case) that routes and handles requests. As opposed to annotation-based endpoint controllers that rely on declaring Intent and using callbacks to complete an activity, the reactive model based on functional endpoints allows request handling to be fully controlled by the application.

2.3. CREATING A REACTIVE SPRING BOOT HTTP SERVICE WITH WEBFLUX

Create a basic reactive Hello World HTTP web service using Spring Boot and WebFlux.

Prerequisites

- JDK 8 or JDK 11 installed
- Maven installed
- A Maven-based application project configured to use Spring Boot

Procedure

1. Add **vertx-spring-boot-starter-http** as a dependency in the **pom.xml** file of your project.

   pom.xml

```
<project>
...
  <dependencies>
   ...
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-http</artifactId>
    </dependency>
   ...
  <dependencies>
...
</project>
```

2. Create a main class for your application and define the router and handler methods.

   HttpSampleApplication.java

```
package dev.snowdrop.vertx.sample.http;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;

import static org.springframework.web.reactive.function.BodyInserters.fromObject;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;
import static org.springframework.web.reactive.function.server.ServerResponse.ok;

@SpringBootApplication
public class HttpSampleApplication {

    public static void main(String[] args) {
```

```
        SpringApplication.run(HttpSampleApplication.class, args);
    }

    @Bean
    public RouterFunction<ServerResponse> helloRouter() {
        return route()
            .GET("/hello", this::helloHandler)
            .build();
    }

    private Mono<ServerResponse> helloHandler(ServerRequest request) {
        String name = request
            .queryParam("name")
            .orElse("World");
        String message = String.format("Hello, %s!", name);

        return ok()
            .body(fromObject(message));
    }
}
```

3. OPTIONAL: Run and test your application locally:

    a. Navigate to the root directory of your Maven project:

    ```
    $ cd myApp
    ```

    b. Package your application:

    ```
    $ mvn clean package
    ```

    c. Start your application from the command line:

    ```
    $ java -jar target/vertx-spring-boot-sample-http.jar
    ```

    d. In a new terminal window, issue an HTTP request on the /**hello** endpoint:

    ```
    $ curl localhost:8080/hello
    Hello, World!
    ```

    e. Provide a custom name with your request to get a personalized response:

    ```
    $ curl http://localhost:8080/hello?name=John
    Hello, John!
    ```

Additional resources

- You can deploy your application to an OpenShift cluster .
- You can also configure your application for deployment on stand-alone Red Hat Enterprise Linux.
- For more detail on creating reactive web services with Spring Boot, see the reactive REST service development guide in the Spring community documentation.

2.4. USING BASIC AUTHENTICATION IN A REACTIVE SPRING BOOT WEBFLUX APPLICATION.

Create a reactive Hello World HTTP web service with basic form-based authentication using Spring Security and WebFlux starters.

Prerequisites

- JDK 8 or JDK 11 installed
- Maven installed
- A Maven-based application project configured to use Spring Boot

Procedure

1. Add **vertx-spring-boot-starter-http** and **spring-boot-starter-security** as dependencies in the **pom.xml** file of your project.

   pom.xml

```
<project>
...
  <dependencies>
   ...
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-http</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
   ...
  <dependencies>
...
</project>
```

2. Create an endpoint controller class for your application:

   HelloController.java

```
package dev.snowdrop.vertx.sample.http.security;

import java.security.Principal;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Mono;

@RestController
public class HelloController {

    @GetMapping("/")
```

```
    public Mono<String> hello(Mono<Principal> principal) {
        return principal
            .map(Principal::getName)
            .map(this::helloMessage);
    }

    private String helloMessage(String username) {
        return "Hello, " + username + "!";
    }
}
```

3. Create the main class of your application:

HttpSecuritySampleApplication.java

```
package dev.snowdrop.vertx.sample.http.security;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HttpSecuritySampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(HttpSecuritySampleApplication.class, args);
    }
}
```

4. Create a **SecurityConfiguration** class that stores the user credentials for accessing the **/hello** endpoint.

SecurityConfiguration.java

```
package dev.snowdrop.vertx.sample.http.security;

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;
import org.springframework.security.core.userdetails.MapReactiveUserDetailsService;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;

@EnableWebFluxSecurity
public class SecurityConfiguration {

    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();

        return new MapReactiveUserDetailsService(user);
    }
}
```

5. OPTIONAL: Run and test your application locally:

   a. Navigate to the root directory of your Maven project:

   ```
   $ cd myApp
   ```

   b. Package your application:

   ```
   $ mvn clean package
   ```

   c. Start your application from the command line:

   ```
   $ java -jar target/vertx-spring-boot-sample-http-security.jar
   ```

   d. Navigate to **http://localhost:8080** using a browser to access the login screen.

   e. Log in using the credentials below:

   - username: user
   - password: user

   You receive a customized greeting when you are logged in:

   ```
   Hello, user!
   ```

   f. Navigate to **http://localhost:8080/logout** using a web browser and use the *Log out* button to log out of your application.

   g. Alternatively, use a terminal to make an unauthenticated HTTP request on **localhost:8080**. You receive HTTP **401 Unauthorized** response from your application.

   ```
   $ curl -I http://localhost:8080
   HTTP/1.1 401 Unauthorized
   WWW-Authenticate: Basic realm="Realm"
   Cache-Control: no-cache, no-store, max-age=0, must-revalidate
   Pragma: no-cache
   Expires: 0
   X-Content-Type-Options: nosniff
   X-Frame-Options: DENY
   X-XSS-Protection: 1 ; mode=block
   Referrer-Policy: no-referrer
   ```

   h. Issue an authenticated request using the example user credentials. You receive a personalized response.

   ```
   $ curl -u user:user http://localhost:8080
   Hello, user!
   ```

Additional resources

- You can deploy your application to an OpenShift cluster .

- You can also configure your application for deployment on stand-alone Red Hat Enterprise Linux.
- For the full specification of the Basic HTTP authentication scheme, see document RFC-7617.
- For the full specification of HTTP authentication extensions for interactive clients, including form-based authentication, see document RFC-8053.

2.5. USING OAUTH2 AUTHENTICATION IN A REACTIVE SPRING BOOT APPLICATION.

Set up OAuth2 authentication for your reactive Spring Boot application and authenticate using your client ID and client secret.

Prerequisites

- JDK 8 or JDK 11 installed
- Maven installed
- A Maven-based application project configured to use Spring Boot
- A GitHub account

Procedure

1. Register a new OAuth 2 application on your Github account. Ensure that you provide the following values in the registration form:
   - Homepage URL: **http://localhost:8080**
   - Authorization callback URL: **http://localhost:8080/login/oauth2/code/github**
     Ensure that you save the client ID and a client secret that you receive upon completing the registration.
2. Add the following dependencies in the **pom.xml** file of your project:
   - **vertx-spring-boot-starter-http**
   - **spring-boot-starter-security**
   - **spring-boot-starter-oauth2-client**
   - **reactor-netty**
     Note that the **reactor-netty** client is required to ensure that **spring-boot-starter-oauth2-client** works properly.

     pom.xml

```
<project>
...
 <dependencies>
 ...
  <dependency>
   <groupId>dev.snowdrop</groupId>
   <artifactId>vertx-spring-boot-starter-http</artifactId>
```

```
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>
    <!-- Spring OAuth2 client only works with Reactor Netty client -->
    <dependency>
      <groupId>io.projectreactor.netty</groupId>
      <artifactId>reactor-netty</artifactId>
    </dependency>
  ...
  <dependencies>
...
</project>
```

3. Create an endpoint controller class for your application:

HelloController.java

```
package dev.snowdrop.vertx.sample.http.oauth;

import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.oauth2.core.user.OAuth2User;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Mono;

@RestController
public class HelloController {

    @GetMapping
    public Mono<String> hello(@AuthenticationPrincipal OAuth2User oauth2User) {
        return Mono.just("Hello, " + oauth2User.getAttributes().get("name") + "!");
    }
}
```

4. Create the main class of your application:

OAuthSampleApplication.java

```
package dev.snowdrop.vertx.sample.http.oauth;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class OAuthSampleApplication {

    public static void main(String[] args) {
```

```
        SpringApplication.run(OAuthSampleApplication.class, args);
    }
}
```

5. Create a YAML configuration file to store the OAuth2 client ID and client secret you received from GitHub upon registering your application.

   src/main/resources/application.yml

```
spring:
  security:
    oauth2:
      client:
        registration:
          github:
            client-id: YOUR_GITHUB_CLIENT_ID
            client-secret: YOUR_GITHUB_CLIENT_SECRET
```

6. OPTIONAL: Run and test your application locally:

   a. Navigate to the root directory of your Maven project:

```
$ cd myApp
```

   b. Package your application:

```
$ mvn clean package
```

   c. Start your application from the command line:

```
$ java -jar target/vertx-spring-boot-sample-http-oauth.jar
```

   d. Navigate to **http://localhost:8080** using a web browser. You are redirected to an OAuth2 application authorization screen on GitHub. If prompted, log in using your GitHub account credentials.

   e. Click *Authorize* to confirm. You are redirected to a screen showing a personalized greeting message.

Additional resources

- You can deploy your application to an OpenShift cluster .
- You can also configure your application for deployment on stand-alone Red Hat Enterprise Linux.
- For more information, see the OAuth2 tutorial in the Spring community documentation. Alternatively, see the tutorial on using OAuth2 with Spring Security.
- For the full OAuth2 authentication framework specification, see document RFC-6749.

2.6. CREATING A REACTIVE SPRING BOOT SMTP MAIL APPLICATION

Create a reactive SMTP email service with Spring Boot with Eclipse Vert.x.

Prerequisites

- JDK 8 or JDK 11 installed
- Maven installed
- A Maven-based application project configured to use Spring Boot
- A SMTP mail server configured on your machine

Procedure

1. Add **vertx-spring-boot-starter-http** and **vertx-spring-boot-starter-mail** as dependencies in the **pom.xml** file of your project.

   pom.xml

```
<project>
...
  <dependencies>
  ...
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-http</artifactId>
    </dependency>
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-mail</artifactId>
    </dependency>
  ...
  <dependencies>
...
</project>
```

2. Create a mail handler class for your application:

   MailHandler.java

```
package dev.snowdrop.vertx.sample.mail;

import dev.snowdrop.vertx.mail.MailClient;
import dev.snowdrop.vertx.mail.MailMessage;
import dev.snowdrop.vertx.mail.SimpleMailMessage;
import org.springframework.stereotype.Component;
import org.springframework.util.MultiValueMap;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;

import static org.springframework.web.reactive.function.server.ServerResponse.noContent;

@Component
public class MailHandler {

    private final MailClient mailClient;
```

```
    public MailHandler(MailClient mailClient) {
        this.mailClient = mailClient;
    }

    public Mono<ServerResponse> send(ServerRequest request) {
        return request.formData()
            .log()
            .map(this::formToMessage)
            .flatMap(mailClient::send)
            .flatMap(result -> noContent().build());
    }

    private MailMessage formToMessage(MultiValueMap<String, String> form) {
        return new SimpleMailMessage()
            .setFrom(form.getFirst("from"))
            .setTo(form.get("to"))
            .setSubject(form.getFirst("subject"))
            .setText(form.getFirst("text"));
    }

}
```

3. Create the main class of your application:

MailSampleApplication.java

```
package dev.snowdrop.vertx.sample.mail;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static org.springframework.http.MediaType.APPLICATION_FORM_URLENCODED;
import static org.springframework.web.reactive.function.server.RequestPredicates.accept;
import static org.springframework.web.reactive.function.server.RouterFunctions.resources;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;

@SpringBootApplication
public class MailSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(MailSampleApplication.class, args);
    }

    @Bean
    public RouterFunction<ServerResponse> mailRouter(MailHandler mailHandler) {
        return route()
                .POST("/mail", accept(APPLICATION_FORM_URLENCODED),
mailHandler::send)
                .build();
    }
```

```
    @Bean
    public RouterFunction<ServerResponse> staticResourceRouter() {
        return resources("/**", new ClassPathResource("static/"));
    }

}
```

4. Create an **application.properties** file to store your SMTP server credentials:

   application.properties

```
vertx.mail.host=YOUR_SMTP_SERVER_HOSTNAME
vertx.mail.username=YOUR_SMTP_SERVER_USERNAME
vertx.mail.password=YOUR_SMTP_SERVER_PASSWORD
```

5. Create a **src/main/resources/static/index.html** file that serves as the frontend of your application. Alternatively, use the example HTML email form available for this procedure.

6. OPTIONAL: Run and test your application locally:

   a. Navigate to the root directory of your Maven project:

```
$ cd myApp
```

   b. Package your application:

```
$ mvn clean package
```

   c. Start your application from the command line.

```
$ java -jar target/vertx-spring-boot-sample-mail.jar
```

   d. Navigate to **http://localhost:8080/index.html** using a web browser to access the email form.

Additional resources

- For more information on setting up an SMTP mail server on RHEL 7, see the Mail Transport Agent Configuration section in the RHEL 7 documentation.
- You can deploy your application to an OpenShift cluster .
- You can also configure your application for deployment on stand-alone Red Hat Enterprise Linux.

2.7. SERVER-SENT EVENTS

Server-sent events (SSE) is a push technology allowing HTTP sever to send unidirectional updates to the client. SSE works by establishing a connection between the event source and the client. The event source uses this connection to push events to the client-side. After the server pushes the events, the connection remains open and can be used to push subsequent events. When the client terminates the request on the server, the connection is closed. SSE represents a more resource-efficient alternative to

polling, where a new connection must be established each time the client polls the event source for updates. As opposed to WebSockets, SSE pushes events in one direction only (that is, from the source to the client). It does not handle bidirectional communication between the event source and the client.

The specification for SSE is incorporated into HTML5, and is widely supported by web browsers, including their legacy versions. SSE can be used from the command line, and is relatively simple to set up compared to other protocols.

SSE is suitable for use cases that require frequent updates from the server to the client, while updates from the client side to the server are expected to be less frequent. Updates form the client side to the server can then be handled over a different protocol, such as REST. Examples of such use cases include social media feed updates or notifications sent to a client when new files are uploaded to a file server.

2.8. USING SERVER-SENT EVENTS IN A REACTIVE SPRING BOOT APPLICATION

Create a simple service that accepts HTTP requests and returns a stream of server-sent events (SSE). When the client establishes a connection to the server and the streaming starts, the connection remains open. The server re-uses the connection to continuously push new events to the client. Canceling the request closes the connection and stops the stream, causing the client to stop receiving updates form the server.

Prerequisites

- JDK 8 or JDK 11 installed
- Maven installed
- A Maven-based application project configured to use Spring Boot

Procedure

1. Add **vertx-spring-boot-starter-http** as a dependency in the **pom.xml** file of your project.

   pom.xml

   ```
   <project>
   ...
     <dependencies>
     ...
       <dependency>
         <groupId>dev.snowdrop</groupId>
         <artifactId>vertx-spring-boot-starter-http</artifactId>
       </dependency>
     ...
     <dependencies>
   ...
   </project>
   ```

2. Create the main class of your application:

   SseExampleApplication.java

   ```
   package dev.snowdrop.vertx.sample.sse;
   ```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SseSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(SseSampleApplication.class, args);
    }
}
```

3. Create a Server-sent Event controller class for your application. In this example, the class generates a stream of random integers and prints them to a terminal application.

SseController.java

```
package dev.snowdrop.vertx.sample.sse;

import java.time.Duration;
import java.util.Random;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;

@RestController
public class SseController {

    @GetMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<Integer> getRandomNumberStream() {
        Random random = new Random();

        return Flux.interval(Duration.ofSeconds(1))
            .map(i -> random.nextInt())
            .log();
    }
}
```

4. OPTIONAL: Run and test your application locally:

   a. Navigate to the root directory of your Maven project:

   ```
   $ cd myApp
   ```

   b. Package your application:

   ```
   $ mvn clean package
   ```

   c. Start your application from the command line:

   ```
   $ java -jar target/vertx-spring-boot-sample-sse.jar
   ```

d. In a new terminal window, issue a HTTP request to **localhost**. You start receiving a continuous stream of random integers from the server-sent event controller:

```
$ curl localhost:8080
data:-2126721954

data:-573499422

data:1404187823

data:1338766210

data:-666543077
...
```

Press **Ctrl**+**C** to cancel your HTTP request and terminate the stream of responses.

Additional resources

- You can deploy your application to an OpenShift cluster .
- You can also configure your application for deployment on stand-alone Red Hat Enterprise Linux.

2.9. WEBSOCKET PROTOCOL

The WebSocket protocol upgrades a standard HTTP connection to make it persistent and subsequently uses that connection to pass specially formatted messages between the client and server of your application. While the protocol relies on HTTP like handshakes to establish the initial connection between client and server over TCP, it uses a special message format for communication between client and server.

Unlike a standard HTTP connection, a WebSocket connection:

- can be used to send messages in both directions
- remains open after the initial request is completed,
- uses special framing headers in messages, which allows you to send non-HTTP-formatted message payloads (for example control data) inside an HTTP request.

As a result, the WebSockets protocol extends the possibilities of a standard HTTP connection while requiring fewer networking resources and decreasing the risk of services failing due to network timeouts (compared to alternative methods of providing a real time messaging functionality, such as HTTP Long Polling).

WebSockets connections are supported by default on most currently available web browsers across different operating systems and hardware architectures, which makes WebSockets a suitable choice for writing cross-platform web-based applications that you can connect to using only a web browser.

2.10. USING WEBSOCKETS IN A REACTIVE APPLICATION BASED ON WEBFLUX

The following example demonstrates how you can use the WebSocket protocol in an application that provides a backend service that you can connect to using a web browser. When you access the web front

end URL of your application using a web browser, the front-end initiates a WebSocket connection to a backend service. You can use the web form available on the website to send values formatted as text strings to the back-end service using the WebSocket connection. The application processes the received value by converting all characters to uppercase and sends the result to the front end using the same WebSocket connection.

Create an application using Spring on Reactive Stack that consists of:

- a back end Java-based service with a WebSocket handler
- a web front end based on HTML and JavaScript.

Prerequisites

- A Maven-based Java application project that uses Spring Boot
- JDK 8 or JDK 11 installed
- Maven installed

Procedure:

1. Add the **vertx-spring-boot-starter-http** as a dependency in the **pom.xml** file of your application project:

   pom.xml

```
...
<dependencies>
  ...
   <dependency>
     <groupId>dev.snowdrop</groupId>
     <artifactId>vertx-spring-boot-starter-http</artifactId>
   </dependency>
  ...
</dependencies>
...
```

2. Create the class file containing the back-end application code:

   /src/main/java/webSocketSampleApplication.java

```
package dev.snowdrop.WebSocketSampleApplication;

import java.util.Collections;
import java.util.Map;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.reactive.HandlerMapping;
import org.springframework.web.reactive.handler.SimpleUrlHandlerMapping;
import org.springframework.web.reactive.socket.WebSocketHandler;
import org.springframework.web.reactive.socket.WebSocketMessage;
import org.springframework.web.reactive.socket.WebSocketSession;
```

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

@SpringBootApplication
public class WebSocketSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebSocketSampleApplication.class, args);
    }

    @Bean
    public HandlerMapping handlerMapping() {
        // Define URL mapping for the socket handlers
        Map<String, WebSocketHandler> handlers = Collections.singletonMap("/echo-upper",
this::toUppercaseHandler);

        SimpleUrlHandlerMapping handlerMapping = new SimpleUrlHandlerMapping();
        handlerMapping.setUrlMap(handlers);
        // Set a higher precedence than annotated controllers (smaller value means higher
precedence)
        handlerMapping.setOrder(-1);

        return handlerMapping;
    }

    private Mono<Void> toUppercaseHandler(WebSocketSession session) {
        Flux<WebSocketMessage> messages = session.receive() // Get incoming messages
stream
            .filter(message -> message.getType() == WebSocketMessage.Type.TEXT) // Filter
out non-text messages
            .map(message -> message.getPayloadAsText().toUpperCase()) // Execute service
logic
            .map(session::textMessage); // Create a response message

        return session.send(messages); // Send response messages
    }
}
```

3. Create the HTML document that serves as a front end for the application. Note, that in the following example, the **<script>** element contains the JavaScript code that handles the communication with the back end of your application:

/src/main/resources/static/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>WebSocket Example</title>
  <script>
  const socket = new WebSocket("ws://localhost:8080/echo-upper");

  socket.onmessage = function(e) {
    console.log("Received a value: " + e.data);
    const messages = document.getElementById("messages");
```

```
    const message = document.createElement("li");
    message.innerHTML = e.data;
    messages.append(message);
  }

  window.onbeforeunload = function(e) {
    console.log("Closing socket");
    socket.close();
  }

  function send(event) {
    event.preventDefault();

    const value = document.getElementById("value-to-send").value.trim();
    if (value.length > 0) {
      console.log("Sending value to socket: " + value);
      socket.send(value);
    }
  }
  </script>
</head>
<body>
<div>
  <h1>Vert.x Spring Boot WebSocket example</h1>
  <p>
    Enter a value to the form below and click submit. The value will be sent via socket to a
backend service.
    The service will then uppercase the value and send it back via the same socket.
  </p>
</div>
<div>
  <form onsubmit="send(event)">
    <input type="text" id="value-to-send" placeholder="A value to be sent"/>
    <input type="submit"/>
  </form>
</div>
<div>
  <ol id="messages"></ol>
</div>
</body>
</html>
```

4. OPTIONAL: Run and test your application locally:

   a. Navigate to the root directory of your Maven project:

      ```
      $ cd myApp
      ```

   b. Package your application:

      ```
      $ mvn clean package
      ```

   c. Start your application from the command line:

      ```
      $ java -jar target/vertx-spring-boot-sample-websocket.jar
      ```

d. Navigate to **http://localhost:8080/index.html** using a web browser. The website shows a web interface that contains

- an input text box,
- a list of processed results,
- a *Submit* button.

5. Enter a string value into the text box and select *Submit*.
6. View the resulting value rendered in uppercase in the list below the input text box.

Additional resources

- You can deploy your application to an OpenShift cluster .
- You can also configure your application for deployment on stand-alone Red Hat Enterprise Linux.

2.11. ADVANCED MESSAGE QUEUING PROTOCOL

The Advanced Message Queuing Protocol (AMQP) is a communication protocol designed to move messages between applications in a a non-blocking way. Standardized as AMQP 1.0, the protocol provides interoperability and messaging integration between new and legacy applications across different network topologies and environments. AMQP works with multiple broker architectures and provides a range of ways to deliver, receive, queue and route messages. AMQP can also work peer-to-peer when you are not using a broker. In a hybrid cloud environment, you can use AMQP to integrate your services with legacy applications without having to deal with processing a variety of different message formats. AMQP Supports real-time asynchronous message processing capabilities and is therefore suitable for use in reactive applications.

2.12. HOW THE AMQP REACTIVE EXAMPLE WORKS

The messaging integration pattern that this example features is a Publisher-Subscriber pattern that 2 queues and a broker.

- The Request queue stores HTTP requests containing strings that you enter using the Web interface to be processed by the text string processor.
- The Result queue stores responses containing the strings that have been converted to Uppercase and are ready to be displayed.

The components that the application consist of are:

- A front-end service that you can use to submit a text string to the application.
- A back-end service that converts the string to uppercase characters.
- A HTTP controller that is configured and provided by the Spring Boot HTTP Starter
- An embedded Artemis AMQP Broker instance that routes messages between 2 messaging queues:

The request queue passes messages containing text strings from the front end to the text string processor service. When you submit a string for processing:

1. The front end service sends a HTTP **POST** request containing your string as the payload of the request to the HTTP controller.

2. The request is picked up by the messaging manager that routes the message to the AMQP Broker.

3. The broker routes the message to the text string processor service. If the text processor service is unavailable to pick up the request, the broker routes the message to the next available processor instance, if such instance is available. Alternatively, the broker waits before resending the request to the same instance when it becomes available again.

4. The text string processor service picks up the message and converts the characters in the string to uppercase. The processor service sends a request with the processed result in uppercase to the AMQP Broker.

5. The AMQP broker routes the request with the processed results to the messaging manager.

6. The messaging manager stores the request with the processed results in the outgoing queue where it can be accessed by the front end service.

The response queue stores HTTP responses that contain results processed by the string processor service. The front end application polls this queue at regular intervals to retrieve the results. When the processed result is ready to be displayed:

1. The front end service sends a HTTP **GET** request to the HTTP controller provided by the Spring Boot HTTP Starter.

2. The HTTP controller routes the request to the messaging manager.

3. When a request previously submitted by the front end for processing is ready and available in the outgoing queue, the messaging manager sends the result as a response to the HTTP **GET** request back to the HTTP controller

4. The HTTP controller routes the response back to the front end service that displays the result.

2.13. USING AMQP IN A REACTIVE APPLICATION

Develop a simple messaging reactive application using the AMQP Client Starter with a Spring Boot HTTP controller. This example application integrates 2 services in a Publisher-Subscriber messaging integration pattern that uses 2 messaging queues and a broker.

This example shows how you can create a basic application with Spring Boot and Eclipse Vert.x on Reactor Netty that consists of 2 services integrated using AMQP messaging. The application consist of the following components:

- A front-end service that you can use to submit text strings to the application
- A back-end service that converts strings to uppercase characters
- An Artemis AMQP broker that routes massages between the services and manages the request queue and response queue.
- A HTTP controller provided by the Spring Boot HTTP Starter

Prerequisites

- A Maven-based Java application project configured to use Spring Boot
- JDK 8 or JDK 11 installed
- Maven installed

Procedure

1. Add the following dependencies to the **pom.xml** file of your application project:

   pom.xml

```
...
<dependencies>
  ...
  <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>vertx-spring-boot-starter-http</artifactId>
  </dependency>
  <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>vertx-spring-boot-starter-amqp</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-artemis</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-jms-server</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-amqp-protocol</artifactId>
   <exclusions>
    <exclusion>
      <groupId>org.apache.qpid</groupId>
      <artifactId>proton-j</artifactId>
    </exclusion>
   </exclusions>
  </dependency>
  ...
</dependencies>
...
```

2. Create the main class file of the example application. This class contains methods that define the respective processing queues for requests and results:

   /src/main/java/AmqpExampleApplication.java

```
package dev.snowdrop.AmqpExampleApplication.java;

import java.util.HashMap;
import java.util.Map;
```

```
import dev.snowdrop.vertx.amqp.AmqpProperties;
import org.apache.activemq.artemis.api.core.TransportConfiguration;
import org.apache.activemq.artemis.core.remoting.impl.netty.NettyAcceptorFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jms.artemis.ArtemisConfigurationCustomizer;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class AmqpExampleApplication {

    final static String PROCESSING_REQUESTS_QUEUE = "processing-requests";

    final static String PROCESSING_RESULTS_QUEUE = "processing-results";

    public static void main(String[] args) {
        SpringApplication.run(AmqpExampleApplication.class, args);
    }

    /**
     * Add Netty acceptor to the embedded Artemis server.
     */
    @Bean
    public ArtemisConfigurationCustomizer artemisConfigurationCustomizer(AmqpProperties
properties) {
        Map<String, Object> params = new HashMap<>();
        params.put("host", properties.getHost());
        params.put("port", properties.getPort());

        return configuration -> configuration
            .addAcceptorConfiguration(new
TransportConfiguration(NettyAcceptorFactory.class.getName(), params));
    }
}
```

3. Create the class file containing the code for the HTTP REST controller that manages the request queue and the response queue by exposing REST endpoints that handle your GET and POST requests:

/src/main/java/Controller.java

```
package dev.snowdrop.vertx.sample.amqp;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.TEXT_EVENT_STREAM_VALUE;

/**
 * Rest controller exposing GET and POST resources to receive processed messages and
submit messages for processing.
 */
```

```
@RestController
public class Controller {

    private final MessagesManager messagesManager;

    public Controller(MessagesManager messagesManager) {
        this.messagesManager = messagesManager;
    }

    /**
     * Get a flux of messages processed up to this point.
     */
    @GetMapping(produces = TEXT_EVENT_STREAM_VALUE)
    public Flux<String> getProcessedMessages() {
        return Flux.fromIterable(messagesManager.getProcessedMessages());
    }

    /**
     * Submit a message for processing by publishing it to a processing requests queue.
     */
    @PostMapping
    public Mono<Void> submitMessageForProcessing(@RequestBody String body) {
        return messagesManager.processMessage(body.trim());
    }
}
```

4. Create the class file containing the messaging manager. The manager controls how applications components publish requests to the request queue and subsequently subscribe to the response queue to obtain processed results:

/src/main/java/MessagesManager.java:

```
package dev.snowdrop.vertx.sample.amqp;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

import dev.snowdrop.vertx.amqp.AmqpClient;
import dev.snowdrop.vertx.amqp.AmqpMessage;
import dev.snowdrop.vertx.amqp.AmqpSender;
import org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;
import reactor.core.Disposable;
import reactor.core.publisher.Mono;

import static
dev.snowdrop.vertx.sample.amqp.AmqpSampleApplication.PROCESSING_REQUESTS_QUE
UE;
import static
dev.snowdrop.vertx.sample.amqp.AmqpSampleApplication.PROCESSING_RESULTS_QUEU
E;
```

```
/**
 * Processor client submits messages to the requests queue and subscribes to the results
queue for processed messages.
 */
@Component
public class MessagesManager implements InitializingBean, DisposableBean {

   private final Logger logger = LoggerFactory.getLogger(MessagesManager.class);

   private final List<String> processedMessages = new CopyOnWriteArrayList<>();

   private final AmqpClient client;

   private Disposable receiverDisposer;

   // Injecting EmbeddedActiveMQ to make sure it has started before creating this
component.
   public MessagesManager(AmqpClient client, EmbeddedActiveMQ server) {
      this.client = client;
   }

   /**
    * Create a processed messages receiver and subscribe to its messages publisher.
    */
   @Override
   public void afterPropertiesSet() {
      receiverDisposer = client.createReceiver(PROCESSING_RESULTS_QUEUE)
          .flatMapMany(receiver -> receiver.flux()
              .doOnCancel(() -> receiver.close().block())) // Close the receiver once subscription
is disposed
          .subscribe(this::handleMessage);
   }

   /**
    * Cancel processed messages publisher subscription.
    */
   @Override
   public void destroy() {
      if (receiverDisposer != null) {
         receiverDisposer.dispose();
      }
   }

   /**
    * Get messages which were processed up to this moment.
    *
    * @return List of processed messages.
    */
   public List<String> getProcessedMessages() {
      return processedMessages;
   }

   /**
    * Submit a message for processing by publishing it to a processing requests queue.
    *
    * @param body Message body to be processed.
```

```
     * @return Mono which is completed once the message is sent.
     */
    public Mono<Void> processMessage(String body) {
        logger.info("Sending message '{}' for processing", body);

        AmqpMessage message = AmqpMessage.create()
            .withBody(body)
            .build();

        return client.createSender(PROCESSING_REQUESTS_QUEUE)
            .map(sender -> sender.send(message))
            .flatMap(AmqpSender::close);
    }

    private void handleMessage(AmqpMessage message) {
        String body = message.bodyAsString();

        logger.info("Received processed message '{}'", body);
        processedMessages.add(body);
    }
}
```

5. Create the class file containing the uppercase processor that receives text strings from the request queue and converts them to uppercase characters. The processor subsequently publishes the results to the response queue:

/src/main/java/UppercaseProcessor.java

```
package dev.snowdrop.vertx.sample.amqp;

import dev.snowdrop.vertx.amqp.AmqpClient;
import dev.snowdrop.vertx.amqp.AmqpMessage;
import dev.snowdrop.vertx.amqp.AmqpSender;
import org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;
import reactor.core.Disposable;
import reactor.core.publisher.Mono;

import static
dev.snowdrop.vertx.sample.amqp.AmqpSampleApplication.PROCESSING_REQUESTS_QUE
UE;
import static
dev.snowdrop.vertx.sample.amqp.AmqpSampleApplication.PROCESSING_RESULTS_QUEU
E;

/**
 * Uppercase processor subscribes to the requests queue, converts each received message
to uppercase and send it to the
 * results queue.
 */
@Component
public class UppercaseProcessor implements InitializingBean, DisposableBean {
```

```
    private final Logger logger = LoggerFactory.getLogger(UppercaseProcessor.class);

    private final AmqpClient client;

    private Disposable receiverDisposer;

    // Injecting EmbeddedActiveMQ to make sure it has started before creating this
component.
    public UppercaseProcessor(AmqpClient client, EmbeddedActiveMQ server) {
        this.client = client;
    }

    /**
     * Create a processing requests receiver and subscribe to its messages publisher.
     */
    @Override
    public void afterPropertiesSet() {
        receiverDisposer = client.createReceiver(PROCESSING_REQUESTS_QUEUE)
            .flatMapMany(receiver -> receiver.flux()
                .doOnCancel(() -> receiver.close().block())) // Close the receiver once subscription
is disposed
            .flatMap(this::handleMessage)
            .subscribe();
    }

    /**
     * Cancel processing requests publisher subscription.
     */
    @Override
    public void destroy() {
        if (receiverDisposer != null) {
            receiverDisposer.dispose();
        }
    }

    /**
     * Convert the message body to uppercase and send it to the results queue.
     */
    private Mono<Void> handleMessage(AmqpMessage originalMessage) {
        logger.info("Processing '{}'", originalMessage.bodyAsString());

        AmqpMessage processedMessage = AmqpMessage.create()
            .withBody(originalMessage.bodyAsString().toUpperCase())
            .build();

        return client.createSender(PROCESSING_RESULTS_QUEUE)
            .map(sender -> sender.send(processedMessage))
            .flatMap(AmqpSender::close);
    }
}
```

6. OPTIONAL: Run and test your application locally:

    a. Navigate to the root directory of your Maven project:

```
$ cd myApp
```

b. Package your application:

```
$ mvn clean package
```

c. Start your application from the command line:

```
$ java -jar target/vertx-spring-boot-sample-amqp.jar
```

d. In a new terminal window, send a number of HTTP **POST** request that contain text strings to be processed to **localhost**

```
$ curl -H "Content-Type: text/plain" -d 'Hello, World' -X POST http://localhost:8080
$ curl -H "Content-Type: text/plain" -d 'Hello again' -X POST http://localhost:8080
```

e. Send an HTTP **GET** request to **localhost**. You receive a HTTP response with the strings in uppercase.

```
$ curl http://localhost:8080
HTTP/1.1 200 OK
Content-Type: text/event-stream;charset=UTF-8
transfer-encoding: chunked

data:HELLO, WORLD

data:HELLO AGAIN
```

Additional resources

- You can deploy your application to an OpenShift cluster .
- You can also configure your application for deployment on stand-alone Red Hat Enterprise Linux.

2.14. APACHE KAFKA

Apache Kafka is a scalable messaging integration system that is designed to exchange messages between processes, applications, and services. Kafka is based on clusters with one or more brokers that maintain a set of topics. Essentially, topics are categories that can be defined for every cluster using topic IDs. Each topic contains pieces of data called records that contain information about the events taking place in your application. Applications connected to the system can add records to these topics, or process and reprocess messages added earlier.

The broker is responsible for handling the communication with client applications and for managing the records in the topic. To ensure that no records are lost, the broker tracks all records in a commit log and keeps track of an offset value for each application. The offset is similar to a pointer that indicates the most recently added record.

Applications can pull the latest records from the topic, or they can change the offset to read records that have been added earlier earlier message. This functionality prevents client applications from becoming overwhelmed with incoming requests in case they can not process them in real time. When

this happens, Kafka prevents loss of data by storing records that cannot be processed in real time in the commit log. when the client application is able to catch up with the incoming requests, it resumes processing records in real time

A broker can manage records in multiple topics by sorting them into topic partitions. Apache Kafka replicates these partitions to allow records form a single topic to be handled by multiple brokers in parallel, allowing you to scale the rate at which your applications process records in a topic. The replicated topic partitions (also called followers) are synchronized with the original topic partition (also called a Leader) to avoid redundancy in processing records. New records are committed to the Leader partition, Followers only replicate the changes made to the leader.

2.15. HOW THE APACHE KAFKA REACTIVE EXAMPLE WORKS

This example application is based on a Publisher-Subscriber message streaming pattern implemented using an Apache Kafka. The components that the application consist of are:

- The **KafkaExampleApplication** class that instantiates the log message producer and consumer
- A WebFlux HTTP controller that is configured and provided by the Spring Boot HTTP Starter. The controller provides rest resources used to publish and read messages.
- A **KafkaLogger** class that defines how the producer publishes messages to the **log** topic on Kafka.
- A **KafkaLog** class that displays messages that the example application receives from the **log** topic on Kafka.

Publishing messages:

1. You make an HTTP POST request to the example application with the log message as the payload.
2. The HTTP controller routes the message to the REST endpoint used for publishing messages, and passes the message to the logger instance.
3. The HTTP controller publishes the received message to the **log** topic on Kafka.
4. KafkaLog instance receives the log message from a Kafka topic.

Reading messages:

1. You send a HTTP **GET** request to the example application URL.
2. The controller gets the messages from the **KafkaLog** instance and returns them as the body of the HTTP response.

2.16. USING KAFKA IN A REACTIVE APPLICATION

This example shows how you can create an example messaging application that uses Apache Kafka with Spring Boot and Eclipse Vert.x on Reactor Netty. The application publishes messages to a Kafka topic and then retrieves them and displays them when you send a request.

The Kafka configuration properties for message topics, URLs, and metadata used by the the Kafka cluster are stored in **src/main/resources/application.yml**.

Prerequisites

- A Maven-based Java application project configured to use Spring Boot
- JDK 8 or JDK 11 installed
- Maven installed

Procedure

1. Add the WebFlux HTTP Starter and the Apache Kafka Starter as dependencies in the **pom.xml** file of your application project:

pom.xml

```
...
<dependencies>
  ...
  <!-- Vert.x WebFlux starter used to handle HTTP requests -->
  <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>vertx-spring-boot-starter-http</artifactId>
  </dependency>
  <!-- Vert.x Kafka starter used to send and receive messages to/from Kafka cluster -->
  <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>vertx-spring-boot-starter-kafka</artifactId>
  </dependency>
  ...
</dependencies>
...
```

1. Create the **KafkaLogger** class. This class functions s a producer and sendas messages The **KafkaLogger** class defines how the Producer publishes messages (also called records) to the topic:

/src/main/java/KafkaLogger.java

```
...
final class KafkaLogger {

    private final KafkaProducer<String, String> producer;

    KafkaLogger(KafkaProducer<String, String> producer) {
        this.producer = producer;
    }

    public Mono<Void> logMessage(String body) {
        // Generic key and value types can be inferred if both key and value are used to create a
builder
        ProducerRecord<String, String> record = ProducerRecord.<String,
String>builder(LOG_TOPIC, body).build();

        return producer.send(record)
            .log("Kafka logger producer")
            .then();
```

```
    }
}
...
```

2. Crate **KafkaLog** class. This class functions as the consumer of kafka messages. **KafkaLog** retrieves messages from the topic an displays them in your terminal:

/src/main/java/KafkaLog.java

```
...
final class KafkaLog implements InitializingBean, DisposableBean {

    private final List<String> messages = new CopyOnWriteArrayList<>();

    private final KafkaConsumer<String, String> consumer;

    private Disposable consumerDisposer;

    KafkaLog(KafkaConsumer<String, String> consumer) {
        this.consumer = consumer;
    }

    @Override
    public void afterPropertiesSet() {
        consumerDisposer = consumer.subscribe(LOG_TOPIC)
            .thenMany(consumer.flux())
            .log("Kafka log consumer")
            .map(ConsumerRecord::value)
            .subscribe(messages::add);
    }

    @Override
    public void destroy() {
        if (consumerDisposer != null) {
            consumerDisposer.dispose();
        }
        consumer.unsubscribe()
            .block(Duration.ofSeconds(2));
    }

    public List<String> getMessages() {
        return messages;
    }
}
...
```

3. Create the class file that contains the the HTTP REST controller. The controller that exposes REST resources that your application uses to handle the logging and reading of messages.

/src/main/java/Controller.java

```
package dev.snowdrop.vertx.sample.kafka;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
```

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.TEXT_EVENT_STREAM_VALUE;

/**
 * HTTP controller exposes GET and POST resources to log messages and to receive the
previously logged ones.
 */
@RestController
public class Controller {

    private final KafkaLogger logger;

    private final KafkaLog log;

    public Controller(KafkaLogger logger, KafkaLog log) {
        this.logger = logger;
        this.log = log;
    }

    /**
     * Get a Flux of previously logged messages.
     */
    @GetMapping(produces = TEXT_EVENT_STREAM_VALUE)
    public Flux<String> getMessages() {
        return Flux.fromIterable(log.getMessages());
    }

    /**
     * Log a message.
     */
    @PostMapping
    public Mono<Void> logMessage(@RequestBody String body) {
        return logger.logMessage(body.trim());
    }
}
```

4. Crate the YAML template that contains the URLs that producers and consumers in your Apache Kafka Cluster use to log and read messages. In this example, the consumer and producer on your Apache Kafka Cluster communicate using port **9092** on **localhost** by default. Note, that you must configure the producers and consumers separately, as the following example shows:

/src/main/resources/application.yml

```
vertx:
  kafka:
    producer:
      bootstrap:
        # The producer in your cluster uses this URL to publish messages to the log.
        servers: localhost:9092
      key:
        # This class assigns the mandatory key attribute that is assigned to each message.
        serializer: org.apache.kafka.common.serialization.StringSerializer
```

```
      value:
        # This class assigns the mandatory value attribute that is assigned to each message.
        serializer: org.apache.kafka.common.serialization.StringSerializer
    consumer:
      bootstrap:
        servers: localhost:9092 # The consumer in your cluster uses this URL to read messages from the log.
      group:
        id: log # The consumer group IDs used to define a group of consumers that subscribe to the same topic. In this example, all consumers belong in the same consumer group.
      key:
        deserializer: org.apache.kafka.common.serialization.StringDeserializer # This class generates the mandatory key attribute that is assigned to each message.
      value:
        deserializer: org.apache.kafka.common.serialization.StringDeserializer # This class generates the mandatory value attribute that is assigned to each message.
```

5. OPTIONAL: Run and test your application locally:

   a. Navigate to the root directory of your Maven project:

   ```
   $ cd vertx-spring-boot-sample-kafka
   ```

   b. Package your application:

   ```
   $ mvn clean package
   ```

   c. Start your application from the command line:

   ```
   $ java -jar target/vertx-spring-boot-sample-kafka.jar
   ```

   d. In a new terminal window, send a number of HTTP **POST** request that contain messages formatted as text strings to **localhost**. The messages are all published to the **log** topic.

   ```
   $ curl -H "Content-Type: text/plain" -d 'Hello, World' -X POST http://localhost:8080
   $ curl -H "Content-Type: text/plain" -d 'Hello again' -X POST http://localhost:8080
   ...
   ```

   e. Send an HTTP **GET** request to **localhost**. You receive a HTTP response that contains all the messages in the topic that your consumers subscribe to.

   ```
   $ curl http://localhost:8080
   HTTP/1.1 200 OK
   Content-Type: text/event-stream;charset=UTF-8
   transfer-encoding: chunked

   data:Hello, World

   data:Hello, again
   ...
   ```

Additional resources

- You can deploy your application to an OpenShift cluster .

- You can also configure your application for deployment on stand-alone Red Hat Enterprise Linux.

CHAPTER 3. DEBUGGING YOUR SPRING BOOT-BASED APPLICATION

This sections contains information about debugging your Spring Boot–based application both in local and remote deployments.

3.1. REMOTE DEBUGGING

To remotely debug an application, you must first configure it to start in a debugging mode, and then attach a debugger to it.

3.1.1. Starting your Spring Boot application locally in debugging mode

One of the ways of debugging a Maven-based project is manually launching the application while specifying a debugging port, and subsequently connecting a remote debugger to that port. This method is applicable at least when launching the application manually using the **mvn spring-boot:run** goal.

Prerequisites

- A Maven-based application

Procedure

1. In a console, navigate to the directory with your application.
2. Launch your application and specify the necessary JVM arguments and the debug port using the following syntax:

```
$ mvn spring-boot:run -Drun.jvmArguments="-Xdebug -
Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=$PORT_NUMBER"
```

$PORT_NUMBER is an unused port number of your choice. Remember this number for the remote debugger configuration.

If you want the JVM to pause and wait for remote debugger connection before it starts the application, change **suspend** to **y**.

3.1.2. Starting an uberjar in debugging mode

If you chose to package your application as a Spring Boot uberjar, debug it by executing it with the following parameters.

Prerequisites

- An uberjar with your application

Procedure

1. In a console, navigate to the directory with the uberjar.
2. Execute the uberjar with the following parameters. Ensure that all the parameters are specified before the name of the uberjar on the line.

```
$ java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=$PORT_NUMBER -
jar $UBERJAR_FILENAME
```

$PORT_NUMBER is an unused port number of your choice. Remember this number for the remote debugger configuration.

If you want the JVM to pause and wait for remote debugger connection before it starts the application, change **suspend** to **y**.

3.1.3. Starting your application on OpenShift in debugging mode

To debug your Spring Boot-based application on OpenShift remotely, you must set the **JAVA_DEBUG** environment variable inside the container to **true** and configure port forwarding so that you can connect to your application from a remote debugger.

Prerequisites

- Your application running on OpenShift.
- The **oc** binary installed.
- The ability to execute the **oc port-forward** command in your target OpenShift environment.

Procedure

1. Using the **oc** command, list the available deployment configurations:

```
$ oc get dc
```

2. Set the **JAVA_DEBUG** environment variable in the deployment configuration of your application to **true**, which configures the JVM to open the port number **5005** for debugging. For example:

```
$ oc set env dc/MY_APP_NAME JAVA_DEBUG=true
```

3. Redeploy the application if it is not set to redeploy automatically on configuration change. For example:

```
$ oc rollout latest dc/MY_APP_NAME
```

4. Configure port forwarding from your local machine to the application pod:
   a. List the currently running pods and find one containing your application:

```
$ oc get pod
NAME                    READY     STATUS      RESTARTS   AGE
MY_APP_NAME-3-1xrsp     0/1       Running     0          6s
...
```

   b. Configure port forwarding:

```
$ oc port-forward MY_APP_NAME-3-1xrsp $LOCAL_PORT_NUMBER:5005
```

Here, **$LOCAL_PORT_NUMBER** is an unused port number of your choice on your local machine. Remember this number for the remote debugger configuration.

5. When you are done debugging, unset the **JAVA_DEBUG** environment variable in your application pod. For example:

```
$ oc set env dc/MY_APP_NAME JAVA_DEBUG-
```

Additional resources

You can also set the **JAVA_DEBUG_PORT** environment variable if you want to change the debug port from the default, which is **5005**.

3.1.4. Attaching a remote debugger to the application

When your application is configured for debugging, attach a remote debugger of your choice to it. In this guide, Red Hat CodeReady Studio is covered, but the procedure is similar when using other programs.

Prerequisites

- The application running either locally or on OpenShift, and configured for debugging.
- The port number that your application is listening on for debugging.
- Red Hat CodeReady Studio installed on your machine. You can download it from the Red Hat CodeReady Studio download page.

Procedure

1. Start Red Hat CodeReady Studio.
2. Create a new debug configuration for your application:
   a. Click **Run→Debug Configurations**.
   b. In the list of configurations, double-click **Remote Java application.** This creates a new remote debugging configuration.
   c. Enter a suitable name for the configuration in the **Name** field.
   d. Enter the path to the directory with your application into the **Project** field. You can use the **Browse...** button for convenience.
   e. Set the **Connection Type** field to *Standard (Socket Attach)* if it is not already.
   f. Set the **Port** field to the port number that your application is listening on for debugging.
   g. Click **Apply.**
3. Start debugging by clicking the **Debug** button in the Debug Configurations window.
   To quickly launch your debug configuration after the first time, click **Run→Debug History** and select the configuration from the list.

Additional resources

- Debug an OpenShift Java Application with JBoss Developer Studio on Red Hat Knowledgebase.
  Red Hat CodeReady Studio was previously called JBoss Developer Studio.
- A Debugging Java Applications On OpenShift and Kubernetes article on OpenShift Blog.

3.2. DEBUG LOGGING

3.2.1. Add Spring Boot debug logging

Add debug logging to your application.

Prerequisites

- An application that you want to debug.

Procedure

1. Declare a **org.apache.commons.logging.Log** object using the **org.apache.commons.logging.LogFactory** for the class you want to add logging.

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
...
private static Log logger = LogFactory.getLog(TheClass.class);
```

2. Add debugging statements using **logger.debug("my logging message")**.

   Example logging statement

```
@GET
@Path("/greeting")
@Produces("application/json")
public Greeting greeting(@QueryParam("name") @DefaultValue("World") String name) {
    String message = String.format(properties.getMessage(), name);

    logger.debug("Message: " + message);

    return new Greeting(message);
}
```

3. Add a **logging.level.fully.qualified.name.of.TheClass=DEBUG** in **src/main/resources/application.properties**.
   For example, if you added a logging statement to **dev.snowdrop.example.service.GreetingEndpoint** you would use:

```
logging.level.dev.snowdrop.example.service.GreetingEndpoint=DEBUG
```

   This enables log messages at the **DEBUG** level and above to be shown in the logs for your class.

3.2.2. Accessing Spring Boot debug logs on localhost

Start your application and interact with it to see the debugging statements.

Prerequisites

- An application with debug logging enabled.

Procedure

1. Start your application.

```
$ mvn spring-boot:run
```

2. Test your application to invoke debug logging.

```
$ curl http://localhost:8080/api/greeting?name=Sarah
```

3. View your application logs to see your debug messages.

```
dev.snowdrop.example.service.GreetingEndpoint      : Message: Hello, Sarah!
```

To disable debug logging, remove **logging.level.fully.qualified.name.of.TheClass=DEBUG** from **src/main/resources/application.properties** and restart your application.

3.2.3. Accessing debug logs on OpenShift

Start your application and interact with it to see the debugging statements in OpenShift.

Prerequisites

- The **oc** CLI client installed and authenticated.
- A Maven-based application with debug logging enabled.

Procedure

1. Deploy your application to OpenShift:

```
$ mvn clean package -Popenshift -Ddekorate.deploy=true
```

2. View the logs:
    1. Get the name of the pod with your application:

```
$ oc get pods
```

    2. Start watching the log output:

```
$ oc logs -f pod/MY_APP_NAME-2-aaaaa
```

    Keep the terminal window displaying the log output open so that you can watch the log output.

3. Interact with your application:
    1. Get the route of your application:

```
$ oc get routes
```

2. Make an HTTP request on the **/api/greeting** endpoint of your application:

```
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

4. Return to the window with your pod logs and inspect debug logging messages in the logs.

```
dev.snowdrop.example.service.GreetingEndpoint      : Message: Hello, Sarah!
```

5. To disable debug logging, remove **logging.level.fully.qualified.name.of.TheClass=DEBUG** from **src/main/resources/application.properties** and redeploy your application.

CHAPTER 4. MONITORING YOUR APPLICATION

This section contains information about monitoring your Spring Boot–based application running on OpenShift.

4.1. ACCESSING JVM METRICS FOR YOUR APPLICATION ON OPENSHIFT

4.1.1. Accessing JVM metrics using Jolokia on OpenShift

Jolokia is a built-in lightweight solution for accessing JMX (Java Management Extension) metrics over HTTP on OpenShift. Jolokia allows you to access CPU, storage, and memory usage data collected by JMX over an HTTP bridge. Jolokia uses a REST interface and JSON-formatted message payloads. It is suitable for monitoring cloud applications thanks to its comparably high speed and low resource requirements.

For Java-based applications, the OpenShift Web console provides the integrated *hawt.io* console that collects and displays all relevant metrics output by the JVM running your application.

Prerequistes

- the **oc** client authenticated
- a Java-based application container running in a project on OpenShift
- latest JDK 1.8.0 image

Procedure

1. List the deployment configurations of the pods inside your project and select the one that corresponds to your application.

```
oc get dc
```

```
NAME        REVISION   DESIRED   CURRENT   TRIGGERED BY
MY_APP_NAME  2          1         1         config,image(my-app:6)
...
```

2. Open the YAML deployment template of the pod running your application for editing.

```
oc edit dc/MY_APP_NAME
```

3. Add the following entry to the **ports** section of the template and save your changes:

```
...
spec:
  ...
  ports:
  - containerPort: 8778
    name: jolokia
    protocol: TCP
  ...
...
```

4. Redeploy the pod running your application.

```
oc rollout latest dc/MY_APP_NAME
```

   The pod is redeployed with the updated deployment configuration and exposes the port **8778**.

5. Log into the OpenShift Web console.

6. In the sidebar, navigate to *Applications* > *Pods*, and click on the name of the pod running your application.

7. In the pod details screen, click *Open Java Console* to access the hawt.io console.

Additional resources

- hawt.io documentation

APPENDIX A. THE SOURCE-TO-IMAGE (S2I) BUILD PROCESS

Source-to-Image (S2I) is a build tool for generating reproducible Docker-formatted container images from online SCM repositories with application sources. With S2I builds, you can easily deliver the latest version of your application into production with shorter build times, decreased resource and network usage, improved security, and a number of other advantages. OpenShift supports multiple build strategies and input sources.

For more information, see the Source-to-Image (S2I) Build chapter of the OpenShift Container Platform documentation.

You must provide three elements to the S2I process to assemble the final container image:

- The application sources hosted in an online SCM repository, such as GitHub.
- The S2I Builder image, which serves as the foundation for the assembled image and provides the ecosystem in which your application is running.
- Optionally, you can also provide environment variables and parameters that are used by S2I scripts.

The process injects your application source and dependencies into the Builder image according to instructions specified in the S2I script, and generates a Docker-formatted container image that runs the assembled application. For more information, check the S2I build requirements, build options and how builds work sections of the OpenShift Container Platform documentation.

APPENDIX B. UPDATING THE DEPLOYMENT CONFIGURATION OF AN EXAMPLE APPLICATION

The deployment configuration for an example application contains information related to deploying and running the application in OpenShift, such as route information or readiness probe location. The deployment configuration of an example application is stored in a set of YAML files. For the example applications that use Nodeshift, the YAML files are located in the **.nodeshift** directory.

IMPORTANT

The deployment configuration files used by Nodeshift do not have to be full OpenShift resource definitions. Nodeshift can take the deployment configuration files and add some missing information to create a full OpenShift resource definition. The resource definitions generated by Dekorate are available in the **target/classes/META-INF/dekorate/** directory. The resource definitions generated by Nodeshift are available in the **tmp/nodeshift/resource/** directory.

Prerequisites

- An existing example project.
- The **oc** CLI client installed.

Procedure

1. Edit an existing YAML file or create an additional YAML file with your configuration update.
   - For example, if your example already has a YAML file with a **readinessProbe** configured, you could change the **path** value to a different available path to check for readiness:

```
spec:
  template:
    spec:
      containers:
        readinessProbe:
          httpGet:
            path: /path/to/probe
            port: 8080
            scheme: HTTP
...
```

   - If a **readinessProbe** is not configured in an existing YAML file, you can also create a new YAML file in the same directory with the **readinessProbe** configuration.
2. Deploy the updated version of your example using Maven or npm.
3. Verify that your configuration updates show in the deployed version of your example.

```
$ oc export all --as-template='my-template'

apiVersion: template.openshift.io/v1
kind: Template
metadata:
  creationTimestamp: null
```

```
  name: my-template
objects:
- apiVersion: template.openshift.io/v1
  kind: DeploymentConfig
  ...
  spec:
    ...
    template:
      ...
      spec:
        containers:
          ...
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /path/to/different/probe
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 60
            periodSeconds: 30
            successThreshold: 1
            timeoutSeconds: 1
          ...
```

Additional resources

If you updated the configuration of your application directly using the web-based console or the **oc** CLI client, export and add these changes to your YAML file. Use the **oc export all** command to show the configuration of your deployed application.

APPENDIX C. ADDITIONAL SPRING BOOT RESOURCES

- OpenShift Architecture Overview
- Spring Boot Microservices On Red Hat OpenShift Container Platform 3
- Spring Cloud Kubernetes
- Spring Boot Project
- Spring Framework Project
- OpenShift Spring Boot Lab Microservices

APPENDIX D. APPLICATION DEVELOPMENT RESOURCES

For additional information about application development with OpenShift, see:

- OpenShift Interactive Learning Portal

To reduce network load and shorten the build time of your application, set up a Nexus mirror for Maven on your OpenShift Container Platform:

- Setting Up a Nexus Mirror for Maven

APPENDIX E. PROFICIENCY LEVELS

Each available example teaches concepts that require certain minimum knowledge. This requirement varies by example. The minimum requirements and concepts are organized in several levels of proficiency. In addition to the levels described here, you might need additional information specific to each example.

Foundational

The examples rated at Foundational proficiency generally require no prior knowledge of the subject matter; they provide general awareness and demonstration of key elements, concepts, and terminology. There are no special requirements except those directly mentioned in the description of the example.

Advanced

When using Advanced examples, the assumption is that you are familiar with the common concepts and terminology of the subject area of the example in addition to Kubernetes and OpenShift. You must also be able to perform basic tasks on your own, for example, configuring services and applications, or administering networks. If a service is needed by the example, but configuring it is not in the scope of the example, the assumption is that you have the knowledge to properly configure it, and only the resulting state of the service is described in the documentation.

Expert

Expert examples require the highest level of knowledge of the subject matter. You are expected to perform many tasks based on feature-based documentation and manuals, and the documentation is aimed at most complex scenarios.