Red Hat build of Quarkus 1.3

Configuring Quarkus applications

Last Updated: 2020-08-20

Legal Notice

Abstract

This guide describes how to configure Quarkus applications.

Table of Contents

PREFACE

As an application developer, you can use Red Hat build of Quarkus to create microservices-based applications written in Java that run in serverless and OpenShift environments. These applications have small memory footprints and fast start-up times.

This guide describes how to configure a Quarkus application using the Eclipse MicroProfile Config method or YAML format. The procedures include configuration examples created using the Quarkus **config-quickstart** exercise.

Prerequisites

- OpenJDK (JDK) 11 is installed and the **JAVA_HOME** environment variable specifies the location of the Java SDK. Red Hat build of Open JDK is available from the Software Downloads page in the Red Hat Customer Portal (login required).
- Apache Maven 3.6.2 or higher is installed. Maven is available from the Apache Maven Project website.
- Maven settings configured to use artifacts from the Quarkus Maven repository. For instructions how to configure Maven settings see the Getting started with Red Hat build of Quarkus .

CHAPTER 1. RED HAT BUILD OF QUARKUS CONFIGURATION OPTIONS

Configuration options enable you to change the settings of your application in a single configuration file. Quarkus supports configuration profiles that let you group related properties and switch between profiles as required.

You can use the MicroProfile Config specification from the Eclipse MicroProfile project to inject configuration properties into your application and configure them using a method defined in your code. By default, Quarkus reads properties from the **application.properties** file located in the **src/main/resources** directory.

By adding the **config-yaml** dependency to your project **pom.xml** file you can add your application properties in the **application.yaml** file using the YAML format.

Quarkus can also read application properties from different sources, such as the file system, database, or any source that can be loaded by a Java application.

CHAPTER 2. CREATING THE CONFIGURATION QUICKSTART PROJECT

The **config-quickstart** project lets you get up and running with a simple Quarkus application using Apache Maven and the Quarkus Maven plug-in. The following procedure demonstrates how you can create a Quarkus Maven project.

Procedure

1. In a command terminal, enter the following command to verify that Maven is using JDK 11 and that the Maven version is 3.6.2 or higher:

```
mvn --version
```

2. If the preceding command does not return JDK 11, add the path to JDK 11 to the PATH environment variable and enter the preceding command again.

3. To generate the project, enter the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.4.Final-redhat-00004:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=config-quickstart \
    -DplatformGroupId=com.redhat.quarkus \
    -DplatformVersion=1.3.4.Final-redhat-00004 \
    -DclassName="org.acme.config.GreetingResource" \
    -Dpath="/greeting"
cd config-quickstart
```

This command creates the following elements in the **./config-quickstart** directory:

- The Maven structure
- An **org.acme.config.GreetingResource** resource
- A landing page that is accessible on **http://localhost:8080** after you start the application
- Example **Dockerfile** file in **src/main/docker**
- The application configuration file
- An associated test

NOTE

Alternatively, you can download a Quarkus Maven project to use in this tutorial from the Quarkus quickstart archive or clone the **Quarkus Quickstarts Git** repository. The exercise is located in the **config-quickstart** directory.

CHAPTER 3. GENERATING AN EXAMPLE CONFIGURATION FILE FOR YOUR QUARKUS APPLICATION

You can create an **application.properties.example** file with all of the available configuration values and documentation for the extensions your application is configured to use. You can repeat this procedure after you install a new extension to see what additional configuration options have been added.

Prerequisites

- You have a Quarkus Maven project.

Procedure

- To create an **application.properties.example** file, enter the following command:

```
./mvnw quarkus:generate-config
```

  This command creates the **application.properties.example** file in the **src/main/resources/** directory. The file contains all of the configuration options exposed through the extensions that you installed. These options are commented out and have a default value where applicable.

  The following example shows the HTTP port configuration entry from the **application.properties.example** file:

```
#quarkus.http.port=8080
```

For a complete list of Quarkus configuration options, see Quarkus-All Configuration Options.

CHAPTER 4. INJECTING CONFIGURATION VALUES INTO YOUR QUARKUS APPLICATION

Red Hat build of Quarkus uses the MicroProfile Config feature to inject configuration data into the application. You can access the configuration through context and dependency injection (CDI) or by using a method defined in your code.

You can use the **@ConfigProperty** annotation to map an object property to a key in the MicroProfile ConfigSources file of your application. This procedure shows you how to inject an individual property configuration into a Quarkus **config-quickstart** project.

Prerequisites

- You have created the Quarkus **config-quickstart** project.

Procedure

1. Open the **src/main/resources/application.properties** file.
2. Add configuration properties to your configuration file where **<KEY>** is the property name and **<VALUE>** is the value of the property:

   ```
   <KEY>=<VALUE>
   ```

   The following example shows how to set the values for the **greeting.message** and the **greeting.name** properties in the Quarkus **config-quickstart** project:

   ```
   greeting.message = hello
   greeting.name = quarkus
   ```

   IMPORTANT

   Use **quarkus** as a prefix to Quarkus properties.

3. Review the **GreetingResource.java** file and make sure it includes the following import statements:

   ```
   import org.eclipse.microprofile.config.inject.ConfigProperty;
   import java.util.Optional;
   ```

4. Define the equivalent properties by annotating them with the following syntax:

   ```
   @ConfigProperty(name = "greeting.message") 1
   String message;

   @ConfigProperty(name = "greeting.suffix", defaultValue="!") 2
   String suffix;

   @ConfigProperty(name = "greeting.name")
   Optional<String> name; 3
   ```

1. If you do not provide a value for this property, the application will fail and throw the following exception message:
2. If you do not provide a value for the **greeting.suffix**, Quarkus resolves it to the default value.
3. If the **Optional** parameter does not have a value, it returns no value for **greeting.name**.

NOTE

To inject a configured value, you can use **@ConfigProperty**. The **@Inject** annotation is not necessary for members annotated with **@ConfigProperty**.

5. Edit your **hello** method to return the following message:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String hello() {
    return message + "  " + name.orElse("world") + suffix;
}
```

6. To compile your Quarkus application in development mode, enter the following command from the project directory:

```
./mvnw quarkus:dev
```

7. To verify that the endpoint returns the message, enter the following command in a new terminal window:

```
curl http://localhost:8080/greeting
```

This command returns the following output:

```
hello quarkus!
```

8. To stop the application, press **CTRL+C**

4.1. ANNOTATING A CLASS WITH @CONFIGPROPERTIES

As an alternative to injecting multiple related configuration values individually, you can use the **@io.quarkus.arc.config.ConfigProperties** annotation to group configuration properties. The following procedure demonstrates the use of **@ConfigProperties** annotation on the Quarkus **config-quickstart** project.

Prerequisites

- You have created the Quarkus **config-quickstart** project.

Procedure

1. Review the **GreetingResource.java** file and make sure it includes the following import statements:

```
package org.acme.config;

import java.util.Optional;
import javax.inject.Inject;
```

2. Create a file **GreetingConfiguration.java** in the **src/main/java/org/acme/config** directory.

3. Add the **@ConfigProperties** and **@Optional** imports to the **GreetingConfiguration.java** file:

```
package org.acme.config;

import io.quarkus.arc.config.ConfigProperties;
import java.util.Optional;
```

4. Create a **GreetingConfiguration** class for the **greeting** properties in your **GreetingConfiguration.java** file:

```
@ConfigProperties(prefix = "greeting") (1)
public class GreetingConfiguration {

    private String message;
    private String suffix = "!"; (2)
    private Optional<String> name;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getSuffix() {
        return suffix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }

    public Optional<String> getName() {
        return name;
    }

    public void setName(Optional<String> name) {
        this.name = name;
    }
}
```

(1) **prefix** is optional. If you do not set the prefix, it will be determined by the class name. In this example, it will be **greeting**.

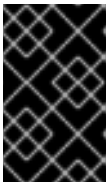
(2) If **greeting.suffix** is not set, **!** will be the default value.

5. Inject the attribute into the **GreetingResource** class using the context and dependency injection (CDI) **@Inject** annotation:

```
@Inject
GreetingConfiguration greetingConfiguration;
```

6. To compile your application in development mode, enter the following command from the project directory:

```
./mvnw quarkus:dev
```



IMPORTANT

If you do not provide values for the class properties, the application fails and a **javax.enterprise.inject.spi.DeploymentException** is thrown indicating a missing value. This does not apply to **Optional** fields and fields with a default value.

4.2. USING NESTED OBJECT CONFIGURATION

You can define a nested class inside an existing class. This procedure demonstrates how to create a nested class configuration in the Quarkus **config-quickstart** project.

Prerequisites

- You have created the Quarkus **config-quickstart** project.

Procedure

1. Review the **GreetingConfiguration.java** file and make sure it includes the following import statements:

```
import io.quarkus.arc.config.ConfigProperties;
import java.util.Optional;
import java.util.List;
```

2. Add the configuration in your **GreetingConfiguration.java** file using the **@ConfigProperties** annotation.
   The following example shows the configuration of the **GreetingConfiguration** class and its properties:

```
@ConfigProperties(prefix = "greeting")
public class GreetingConfiguration {

    public String message;
    public String suffix = "!";
    public Optional<String> name;
}
```

3. Add a nested class configuration similar to the following example:

```
@ConfigProperties(prefix = "greeting")
public class GreetingConfiguration {
```

```
    public String message;
    public String suffix = "!";
    public Optional<String> name;
    public HiddenConfig hidden;

    public static class HiddenConfig {
        public Integer prizeAmount;
        public List<String> recipients;
    }
}
```

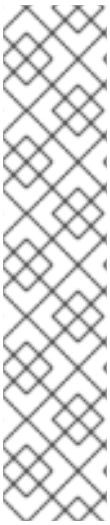
This example shows a nested class **HiddenConfig**. The name of the field, in this case **hidden**, determines the name of the properties bound to the object.

4. Add the equivalent configuration properties to your **application.properties** file.
The following example shows the value of properties for the **GreetingConfiguration** and **HiddenConfig** classes:

```
greeting.message = hello
greeting.name = quarkus
greeting.hidden.prize-amount=10
greeting.hidden.recipients=Jane,John
```

5. To compile your application in development mode, enter the following command from the project directory:

```
./mvnw quarkus:dev
```



NOTE

Classes annotated with **@ConfigProperties** can be annotated with Bean Validation annotations similar to the following example:

```
@ConfigProperties(prefix = "greeting")
public class GreetingConfiguration {

    @Size(min = 20)
    public String message;
    public String suffix = "!";
}
```

Your project must include the **quarkus-hibernate-validator** dependency.

4.3. ANNOTATING AN INTERFACE WITH @CONFIGPROPERTIES

An alternative method for managing properties is to define them as an interface. If you annotate an interface with **@ConfigProperties**, the interface can extend other interfaces, and you can use methods from the entire interface hierarchy to bind properties.

This procedure shows an implementation of the **GreetingConfiguration** class as an interface in the Quarkus **config-quickstart** project.

Prerequisites

- You have created the Quarkus **config-quickstart** project.

Procedure

1. Review the **GreetingConfiguration.java** file and make sure it includes the following import statements:

```
package org.acme.config;

import io.quarkus.arc.config.ConfigProperties;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import java.util.Optional;
```

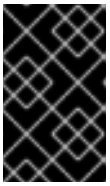
2. Add a **GreetingConfiguration** class as an interface to your **GreetingConfiguration.java** file:

```
@ConfigProperties(prefix = "greeting")
public interface GreetingConfiguration {

    @ConfigProperty(name = "message") 1
    String message();

    @ConfigProperty(defaultValue = "!")
    String getSuffix(); 2

    Optional<String> getName(); 3
}
```

1. You must set the **@ConfigProperty** annotation because the name of the configuration property does not follow the getter method naming conventions.
2. In this example, **name** was not set so the corresponding property will be **greeting.suffix**.
3. You do not need to specify the **@ConfigProperty** annotation because the method name follows the getter method naming conventions (**greeting.name** being the corresponding property) and no default value is needed.

3. To compile your application in development mode, enter the following command from the project directory:

```
./mvnw quarkus:dev
```



IMPORTANT

If you do not provide a value for an interface field, the application fails and an **javax.enterprise.inject.spi.DeploymentException** is thrown indicating a missing value. This does not apply to **Optional** fields and fields with a default value.

CHAPTER 5. ACCESSING THE CONFIGURATION FROM CODE

You can access the configuration by using a method defined in your code. You can achieve dynamic lookups or retrieve configured values from classes that are neither CDI beans or JAX-RS resources.

You can access the configuration using the **org.eclipse.microprofile.config.ConfigProvider.getConfig()** method. The **getValue** method of the **Config object** returns the values of the configuration properties.

Prerequisites

You have a Quarkus Maven project.

Procedure

Access the configuration using one of the following options:

- To access a configuration of a property that is defined already in your **application.properties** file, use the following syntax where **DATABASE.NAME** is the name of a property that is assigned to a **databaseName** variable:

```
String databaseName = ConfigProvider.getConfig().getValue("DATABASE.NAME",
String.class);
```

- To access a configuration of a property that might not be defined in your **application.properties** file, use the following syntax:

```
Optional<String> maybeDatabaseName =
ConfigProvider.getConfig().getOptionalValue("DATABASE.NAME", String.class);
```

CHAPTER 6. SETTING CONFIGURATION PROPERTIES

By default, Quarkus reads properties from the **application.properties** file located in the **src/main/resources** directory. If you change build properties, make sure to repackage your application.

Quarkus configures most properties during build time. Extensions can define properties as overridable at run time, for example the database URL, a user name, and a password which can be specific to your target environment.

Prerequisites

- You have a Quarkus Maven project.

Procedure

1. To package your Quarkus project, enter the following command:

```
./mvnw clean package
```

2. Use one of the following methods to set the configuration properties:

- Setting system properties:
  Enter the following command where **<KEY>** is the name of the configuration property you want to add and **<VALUE>** is the value of the property:

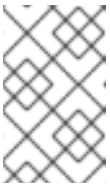
```
java -D<KEY>=<VALUE> -jar target/myapp-runner.jar
```

  For example, to set the value of the **quarkus.datasource.password** property, enter the following command:

```
java -Dquarkus.datasource.password=youshallnotpass -jar target/myapp-runner.jar
```

- Setting environment variables:
  Enter the following command where **<KEY>** is the name of the configuration property you want to set and **<VALUE>** is the value of the property:

```
export <KEY>=<VALUE> ; java -jar target/myapp-runner.jar
```



NOTE

Environment variable names follow the conversion rules of Eclipse MicroProfile. Convert the name to upper case and replace any character that is not alphanumeric with an underscore (_).

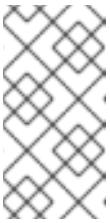
- Using an environment file:
  Create an **.env** file in your current working directory and add configuration properties where **<KEY>** is the property name and **<VALUE>** is the value of the property:

```
<KEY>=<VALUE>
```

NOTE

For development mode, this file can be located in the root directory of your project, but it is advised to not track the file in version control. If you create an **.env** file in the root directory of your project, you can define keys and values that the program reads as properties.

- Using the **application.properties** file.
  Place the configuration file in **$PWD/config/application.properties** directory where the application runs so any runtime properties defined in that file will override the default configuration.



NOTE

You can also use the **config/application.properties** features in development mode. Place the **config/application.properties** inside the **target** directory. Any cleaning operation from the build tool, for example **mvn clean**, will remove the **config** directory as well.

CHAPTER 7. USING CONFIGURATION PROFILES

You can use different configuration profiles depending on your environment. Configuration profiles enable you to have multiple configurations in the same file and select between them using a profile name. Red Hat build of Quarkus has three configuration profiles. In addition, you can create your own custom profiles.

Quarkus default profiles:

- **dev**: Activated in development mode
- **test**: Activated when running tests
- **prod**: The default profile when not running in development or test mode

Prerequisites

- You have a Quarkus Maven project.

Procedure

1. Open your Java resource file and add the following import statement:

```
import io.quarkus.runtime.configuration.ProfileManager;
```

2. To display the current configuration profile, add a log invoking the **ProfileManager.getActiveProfile()** method:

```
LOGGER.infof("The application is starting with profile `%s`",
ProfileManager.getActiveProfile());
```

NOTE

It is not possible to access the current profile using the **@ConfigProperty("quarkus.profile")** method.

7.1. SETTING A CUSTOM CONFIGURATION PROFILE

You can create as many configuration profiles as you want. You can have multiple configurations in the same file and you can select between them using a profile name.

Procedure

1. To set a custom profile, create a configuration property with the profile name in the **application.properties** file, where **<KEY>** is the name of the property, **<VALUE>** is the property value, and **<PROFILE>** is the name of a profile:

```
%<PROFILE>.<KEY>=<VALUE>
```

In the following example configuration, the value of **quarkus.http.port** is 9090 by default, and becomes 8181 when the **dev** profile is activated:

```
quarkus.http.port=9090
%dev.quarkus.http.port=8181
```

2. Use one of the following methods to enable a profile:

   - Set the **quarkus.profile** system property.
     - To enable a profile using the **quarkus.profile** system property, enter the following command:

       ```
       mvn -D<KEY>=<VALUE> quarkus:<PROFILE>
       ```

   - Set the **QUARKUS_PROFILE** environment variable.
     - To enable profile using an environment variable, enter the following command:

       ```
       export QUARKUS_PROFILE=<PROFILE>
       ```

       NOTE

       The system property value takes precedence over the environment variable value.

3. To repackage the application and change the profile, enter the following command:

   ```
   ./mvnw package -Dquarkus.profile=<PROFILE>
   java -jar target/myapp-runner.jar
   ```

   The following example shows a command that activates the **prod-aws** profile:

   ```
   ./mvnw package -Dquarkus.profile=prod-aws
   java -jar target/myapp-runner.jar
   ```

NOTE

The default Quarkus application runtime profile is set to the profile used to build the application. Red Hat build of Quarkus automatically selects a profile depending on your environment mode. For example, when your application is running, Quarkus is in **prod** mode.

CHAPTER 8. SETTING CUSTOM CONFIGURATION SOURCES

By default, Quarkus reads properties from the **application.properties** file. However, because Quarkus supports the MicroProfile Config feature, you can introduce custom configuration sources and load a configuration from another source.

You can introduce custom configuration sources for your configured values by providing classes that implement the **org.eclipse.microprofile.config.spi.ConfigSource** and the **org.eclipse.microprofile.config.spi.ConfigSourceProvider** interfaces. This procedure demonstrates how you can implement a custom configuration source on your Quarkus project.

Prerequisite

You have created the Quarkus **config-quickstart** project.

Procedure

1. Create an **ExampleConfigSourceProvider.java** file within your project and add the following imports:

```
package org.acme.config;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import org.eclipse.microprofile.config.spi.ConfigSource;
import org.eclipse.microprofile.config.spi.ConfigSourceProvider;
```

2. Create a class implementing the **ConfigSourceProvider** interface and make sure to override its **getConfigSources** method to return a list of **ConfigSource** objects:
The following example shows an implementation of a custom **ConfigSourceProvider** and a **ConfigSource** classes:

```
public class ExampleConfigSourceProvider implements ConfigSourceProvider {

    private final int times = 2;
    private final String name = "example";
    private final String value = "value";

    @Override
    public Iterable<ConfigSource> getConfigSources(ClassLoader forClassLoader) {
        InMemoryConfigSource configSource = new
InMemoryConfigSource(Integer.MIN_VALUE, "example config source");
        for (int i = 0; i < this.times; i++) {
            configSource.add(this.name + ".key" + (i + 1), this.value + (i + 1));
        }
        return Collections.singletonList(configSource);
    }

    private static final class InMemoryConfigSource implements ConfigSource {

        private final Map<String, String> values = new HashMap<>();
        private final int ordinal;
```

```
        private final String name;

        private InMemoryConfigSource(int ordinal, String name) {
            this.ordinal = ordinal;
            this.name = name;
        }

        public void add(String key, String value) {
            values.put(key, value);
        }

        @Override
        public Map<String, String> getProperties() {
            return values;
        }

        @Override
        public Set<String> getPropertyNames() {
            return values.keySet();
        }

        @Override
        public int getOrdinal() {
            return ordinal;
        }

        @Override
        public String getValue(String propertyName) {
            return values.get(propertyName);
        }

        @Override
        public String getName() {
            return name;
        }
    }
}
```

3. Create the **org.eclipse.microprofile.config.spi.ConfigSourceProvider** service file in the **META-INF/services/** directory.

4. Open the **org.eclipse.microprofile.config.spi.ConfigSourceProvider** file and add the fully qualified name of your custom ConfigSourceProvider class:

```
org.acme.config.ExampleConfigSourceProvider
```

5. To compile the application in development mode, enter the following command from the project directory:

```
./mvnw quarkus:dev
```

When you restart your application, Quarkus will pick up the custom configuration provider.

CHAPTER 9. USING CUSTOM CONFIGURATION CONVERTERS AS CONFIGURATION VALUES

You can store custom types as configuration values by implementing **org.eclipse.microprofile.config.spi.Converter<T>** and adding its fully qualified class name into the **META-INF/services/org.eclipse.microprofile.config.spi.Converter** file.

Prerequisites

- You have created the Quarkus **config-quickstart** project.

Procedure

1. Include the fully qualified class name of the converter in your **META-INF/services/org.eclipse.microprofile.config.spi.Converter** service file as shown in the following example:

```
org.acme.config.MicroProfileCustomValueConverter
org.acme.config.SomeOtherConverter
org.acme.config.YetAnotherConverter
```

2. Implement the converter class to override the convert method:

```
package org.acme.config;

import org.eclipse.microprofile.config.spi.Converter;

public class MicroProfileCustomValueConverter implements
Converter<MicroProfileCustomValue> {

    @Override
    public MicroProfileCustomValue convert(String value) {
        return new MicroProfileCustomValue(Integer.valueOf(value));
    }
}
```

NOTE

Your custom converter class must be **public** and must have a **public** no-argument constructor. Your custom converter class cannot be **abstract**.

3. Use your custom type as a configuration value:

```
@ConfigProperty(name = "configuration.value.name")
MicroProfileCustomValue value;
```

Additional resources:

- There are converters that convert your property file content from **String** to typed Java types. For more information, see list of converters in the **microprofile-config** GitHub repository.

9.1. SETTING CUSTOM CONVERTERS PRIORITY

The default priority for all Quarkus core converters is 200 and for all other converters it is 100. However, you can set a higher priority for your custom converters using the **javax.annotation.Priority** annotation.

The following procedure demonstrates an implementation of a custom converter **MicroProfileCustomValue** that is assigned a priority of 150 and will take precedence over **MicroProfileCustomValueConverter** which has a value of 100.

Prerequisites

- You have created the Quarkus **config-quickstart** project.

Procedure

1. Add the following import statements to your service file:

```
package org.acme.config;

import javax.annotation.Priority;
import org.eclipse.microprofile.config.spi.Converter;
```

2. Set a priority for your custom converter by annotating the class with the **@Priority** annotation and passing it a priority value:

```
@Priority(150)
public class MyCustomConverter implements Converter<MicroProfileCustomValue> {

    @Override
    public MicroProfileCustomValue convert(String value) {

        final int secretNumber;
        if (value.startsFrom("OBF:")) {
            secretNumber = Integer.valueOf(SecretDecoder.decode(value));
        } else {
            secretNumber = Integer.valueOf(value);
        }

        return new MicroProfileCustomValue(secretNumber);
    }
}
```

NOTE

If you add a new converter, you must list it in the **META-INF/services/org.eclipse.microprofile.config.spi.Converter** service file.

CHAPTER 10. ADDING YAML CONFIGURATION SUPPORT

Red Hat build of Quarkus supports YAML configuration files through the **SmallRye Config** implementation of Eclipse MicroProfile Config. You can add the **Quarkus Config YAML** extension and use YAML over properties for configuration. Quarkus supports using **application.yml** as well as **application.yaml** as the name of the YAML file.

The YAML configuration file takes precedence over the **application.properties** file. The recommended approach is to delete the **application.properties** file and use only one type of configuration file to avoid errors.

Procedure

- Use one of the following methods to add the YAML extension in your project:
  - Open the **pom.xml** file and add the **quarkus-config-yaml** extension as a dependency:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-config-yaml</artifactId>
</dependency>
```

  - To add the **quarkus-config-yaml** extension from the command line, enter the following command from your project directory:

```
./mvnw quarkus:add-extension -Dextensions="quarkus-config-yaml"
```

10.1. USING NESTED OBJECT CONFIGURATION WITH YAML

You can define a nested class inside an already existing class. The following YAML configuration file example shows how you can set nested properties in your Quarkus application using the YAML format.

Prerequisites

- You have an existing Quarkus project that can read YAML configuration files.

Procedure

1. To start Quarkus in development mode, enter the following command in the directory that contains your Quarkus application **pom.xml** file:

```
./mvnw quarkus:dev
```

2. Open your YAML configuration file.
3. Add a nested class configuration similar to the following syntax:

```
# YAML supports comments
quarkus:
  datasource:
    url: jdbc:postgresql://localhost:5432/some-database
    driver: org.postgresql.Driver
    username: quarkus
```

```
    password: quarkus

# REST Client configuration property
org:
  acme:
    restclient:
      CountriesService/mp-rest/url: https://restcountries.eu/rest

# For configuration property names that use quotes, do not split the string inside the quotes.
quarkus:
  log:
    category:
      "io.quarkus.category":
        level: INFO
```

10.2. SETTING CUSTOM CONFIGURATION PROFILES WITH YAML

Red Hat build of Quarkus lets you create profile dependent configurations and switch between them as required.The following procedure demonstrates how you can provide a profile dependent configuration with YAML.

Prerequisites

- You have a Quarkus project configured to read YAML configuration files.

Procedure

1. Open your YAML configuration file.
2. To set a profile dependent configuration, add the profile name before defining the key-value pairs using the **"%profile"** syntax:
   In the following example the PostgreSQL database is configured to be available at the **jdbc:postgresql://localhost:5432/some-database** URL when Quarkus runs in the development mode:

   ```
   "%dev":
     quarkus:
       datasource:
         url: jdbc:postgresql://localhost:5432/some-database
         driver: org.postgresql.Driver
         username: quarkus
         password: quarkus
   ```

3. If you stopped the application, enter the following command to restart it:

   ```
   ./mvnw quarkus:dev
   ```

10.3. MANAGING CONFIGURATION KEY CONFLICTS

Structured formats such as YAML only support a subset of the possible configuration namespace. The following procedure shows a solution of a conflict between two configuration properties, **quarkus.http.cors** and **quarkus.http.cors.methods**, where one property is the prefix of another.

Prerequisites

- You have a Quarkus project configured to read YAML configuration files.

Procedure

1. Open your YAML configuration file.
2. To define a YAML property as a prefix of another property, add a tilde (~) in the scope of the property as shown in the following example:

```
quarkus:
  http:
    cors:
      ~: true
      methods: GET,PUT,POST
```

3. To compile your Quarkus application in development mode, enter the following command from the project directory:

```
./mvnw quarkus:dev
```

NOTE

You can use YAML keys for conflicting configuration keys at any level because they are not included in the assembly of configuration property name.

CHAPTER 11. UPDATING THE FUNCTIONAL TEST TO VALIDATE CONFIGURATION CHANGES

Before you test the functionality of your application, you must update the functional test to reflect the changes you made to the endpoint of your application. The following procedure shows how you can update your **testHelloEndpoint** method on the Quarkus **config-quickstart** project.

Procedure

1. Open the **GreetingResourceTest.java** file.
2. Update the content of the **testHelloEndpoint** method:

```
package org.acme.config;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class GreetingResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
          .when().get("/greeting")
          .then()
             .statusCode(200)
             .body(is("hello quarkus!")); // Modified line
    }

}
```

CHAPTER 12. PACKAGING AND RUNNING YOUR QUARKUS APPLICATION

After you compile your Quarkus project, you can package it in a JAR file and run it from the command line.

Prerequisites

- You have compiled your Quarkus project.

Procedure

1. To package your Quarkus project, enter the following command in the **root** directory:

```
./mvnw clean package
```

   This command produces the following JAR files in the /**target** directory:

   - **config-quickstart-1.0-SNAPSHOT.jar**: Contains the classes and resources of the projects. This is the regular artifact produced by the Maven build.
   - **config-quickstart-1.0-SNAPSHOT-runner.jar**: Is an executable JAR file. Be aware that this file is not an uber-JAR file because the dependencies are copied into the **target/lib** directory.

2. If development mode is running, press **CTRL+C** to stop development mode. If you do not do this, you will have a port conflict.

3. To run the application, enter the following command:

```
java -jar target/config-quickstart-1.0-SNAPSHOT-runner.jar
```

NOTE

The **Class-Path** entry of the **MANIFEST.MF** file from the **runner** JAR file explicitly lists the JAR files from the **lib** directory. If you want to deploy your application from another location, you must copy the **runner** JAR file as well as the **lib** directory.