Red Hat build of Quarkus 3.2

Configuring your Red Hat build of Quarkus applications by using a properties file

Last Updated: 2024-04-04

Legal Notice

Abstract

This guide describes how to configure Red Hat build of Quarkus applications by using a properties file.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. CONFIGURING YOUR RED HAT BUILD OF QUARKUS APPLICATIONS BY USING A PROPERTIES FILE

As an application developer, you can use Red Hat build of Quarkus to create microservices-based applications written in Java that run on OpenShift and serverless environments. Applications compiled to native executables have small memory footprints and fast startup times.

You can configure your Quarkus application by using either of the following methods:

- Setting properties in the **application.properties** file
- Applying structured configuration in YAML format by updating the **application.yaml** file

You can also extend and customize the configuration for your application by doing the following:

- Substituting and composing configuration property values by using property expressions
- Implementing MicroProfile-compliant classes with custom configuration source converters that read configuration values from different external sources
- Using configuration profiles to maintain separate sets of configuration values for your development, test, and production environments

The procedures include configuration examples that are created by using the Quarkus **config-quickstart** exercise.

Prerequisites

- You have installed OpenJDK 11 or 17 and set the **JAVA_HOME** environment variable to specify the location of the Java SDK.
  - To download the Red Hat build of OpenJDK, log in to the Red Hat Customer Portal and go to Software Downloads.
- You have installed Apache Maven 3.8.6 or later.
  - Download Maven from the Apache Maven Project website.
- You have configured Maven to use artifacts from the Quarkus Maven repository.
  - To learn how to configure Maven settings, see Getting started with Quarkus.

1.1. RED HAT CONFIGURATION OPTIONS

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

By default, Quarkus reads properties from the **application.properties** file located in the **src/main/resources** directory. You can also configure Quarkus to read properties from a YAML file instead.

When you add the **quarkus-config-yaml** dependency to your project **pom.xml** file, you can configure and manage your application properties in the **application.yaml** file. For more information, see Adding YAML configuration support.

Red Hat build of Quarkus also supports MicroProfile Config, which enables you to load the configuration of your application from other sources.

You can use the MicroProfile Config specification from the Eclipse MicroProfile project to inject configuration properties into your application and access them using a method defined in your code.

Quarkus can also read application properties from different origins, including the following sources:

- The file system
- A database
- A Kubernetes or OpenShift Container Platform **ConfigMap** or Secret object
- Any source that can be loaded by a Java application

1.2. CREATING THE CONFIGURATION QUICKSTART PROJECT

With the **config-quickstart** project, you can get up and running with a simple Quarkus application by using Apache Maven and the Quarkus Maven plugin. The following procedure describes how you can create a Quarkus Maven project.

Prerequisites

- You have installed OpenJDK 11 or 17 and set the **JAVA_HOME** environment variable to specify the location of the Java SDK.
  - To download Red Hat build of OpenJDK, log in to the Red Hat Customer Portal and go to Software Downloads.
- You have installed Apache Maven 3.8.6 or later.
  - Download Maven from the Apache Maven Project website.

Procedure

1. Verify that Maven is using OpenJDK 11 or 17 and that the Maven version is 3.8.6 or later:

```
mvn --version
```

2. If the **mvn** command does not return OpenJDK 11 or 17, ensure that the directory where OpenJDK 11 or 17 is installed on your system is included in the **PATH** environment variable:

```
export PATH=$PATH:<path_to_JDK>
```

3. Enter the following command to generate the project:

```
mvn com.redhat.quarkus.platform:quarkus-maven-plugin:3.2.11.Final-redhat-00001:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=config-quickstart \
    -DplatformGroupId=com.redhat.quarkus.platform \
    -DplatformVersion=3.2.11.Final-redhat-00001 \
    -DclassName="org.acme.config.GreetingResource" \
    -Dpath="/greeting"
cd config-quickstart
```

Verification

The preceding **mvn** command creates the following items in the **config-quickstart** directory:

- The Maven project directory structure
- An **org.acme.config.GreetingResource** resource
- A landing page that you can access at **http://localhost:8080** after you start the application
- Associated unit tests for testing your application in native mode and JVM mode
- Example **Dockerfile.jvm** and **Dockerfile.native** files in the **src/main/docker** subdirectory
- The application configuration file

NOTE

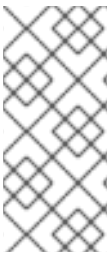
Alternatively, you can download a Quarkus Maven project to use in this tutorial from the Quarkus quickstart archive or clone the **Quarkus Quickstarts** Git repository. The Quarkus **config-quickstart** exercise is located in the **config-quickstart** directory.

1.3. INJECTING CONFIGURATION VALUES INTO YOUR RED HAT BUILD OF QUARKUS APPLICATION

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

Use the **@ConfigProperty** annotation to map an object property to a key in the **MicroProfile Config Sources** file of your application.

The following procedure and examples show how you can inject an individual property configuration into a Quarkus **config-quickstart** project by using the Red Hat build of Quarkus Application configuration file, **application.properties**.



NOTE

You can use a MicroProfile Config configuration file, **src/main/resources/META-INF/microprofile-config.properties**, in exactly the same way you use the **application.properties** file.

Using the **application.properties** file is the preferred method.

Prerequisites

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

Procedure

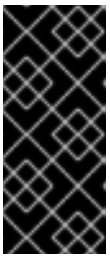
1. Open the **src/main/resources/application.properties** file.
2. Add configuration properties to your configuration file where **<property_name>** is the property name and **<value>** is the value of the property:

```
<property_name>=<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:

Example application.properties file

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



IMPORTANT

When you are configuring your applications, do not prefix application-specific properties with the string **quarkus**. The **quarkus** prefix is reserved for configuring Quarkus at the framework level. Using **quarkus** as a prefix for application-specific properties might lead to unexpected results when your application runs.

3. Review the **GreetingResource.java** Java file in your project. The file contains the **GreetingResource** class with the **hello()** method that returns a message when you send an HTTP request on the **/greeting** endpoint:

Example GreetingResource.java file

```
import java.util.Optional;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import org.eclipse.microprofile.config.inject.ConfigProperty;

@Path("/greeting")
public class GreetingResource {

    String message;
    Optional<String> name;
    String suffix;


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

In the example provided, the values of the **message** and **name** strings in the **hello()** method are not initialized. The application throws a **NullPointerException** when the endpoint is called and starts successfully in this state.

4. Define the **message**, **name**, and **suffix** fields, and annotate them with **@ConfigProperty**, matching the values that you defined for the **greeting.message** and **greeting.name** properties.

Use the **@ConfigProperty** annotation to inject the configuration value for each string. For example:

Example GreetingResource.java file

```
import java.util.Optional;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import org.eclipse.microprofile.config.inject.ConfigProperty;

@Path("/greeting")
public class GreetingResource {

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

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

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

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

1. If you do not configure a value for the **greeting.message** string, the application fails and throws the following exception: **jakarta.enterprise.inject.spi.DeploymentException: io.quarkus.runtime.configuration.ConfigurationException: Failed to load config value of type class java.lang.String for: greeting.message**
2. If you do not configure a value for the **greeting.suffix**, Quarkus resolves it to the default value.
3. If you do not define the **greeting.name** property, the value of **name** is not available. Your application still runs even when this value is not available because you set the **Optional** parameter on **name**.

NOTE

To inject a configured value, you can use **@ConfigProperty**. You do not need to include the **@Inject** annotation for members that you annotate with **@ConfigProperty**.

5. Compile and start your application in development mode:

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

6. Enter the following command in a new terminal window to verify that the endpoint returns the message:

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

This command returns the following output:

```
hello quarkus!
```

7. To stop the application, press Ctrl+C.

1.4. 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 that 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
    }

}
```

1.5. SETTING CONFIGURATION PROPERTIES

By default, Quarkus reads properties from the **application.properties** file that is in the **src/main/resources** directory. If you change build properties, ensure that you repackage your application.

Quarkus configures most properties during build time. Extensions can define properties as overridable at runtime, 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 **<property_name>** is the name of the configuration property that you want to add and **<value>** is the value of the property:

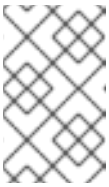
```
java -D<property_name>=<value> -jar target/myapp-runner.jar
```

  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 **<property_name>** is the name of the configuration property that you want to set and **<value>** is the value of the property:

```
export <property_name>=<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 a **.env** file in your current working directory and add configuration properties, where **<PROPERTY_NAME>** is the property name and **<value>** is the value of the property:

```
<PROPERTY_NAME>=<value>
```

NOTE

In development mode, this file is in the root directory of your project. Do not track the file in version control. If you create a **.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 the **$PWD/config/application.properties** directory where the application runs so that any runtime properties that are defined in that file override the default configuration.

NOTE

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

1.6. ADVANCED CONFIGURATION MAPPING

The following advanced mapping procedures are extensions that are specific to Red Hat build of Quarkus and are outside of the MicroProfile Config specification.

1.6.1. Annotating an interface with @ConfigMapping

Instead of individually injecting multiple related configuration values, use the **@io.smallrye.config.ConfigMapping** annotation to group configuration properties. The following procedure describes how you can use the **@ConfigMapping** annotation on the Quarkus **config-quickstart** project.

Prerequisites

- You have created the Quarkus **config-quickstart** project.
- You have defined the **greeting.message** and **greeting.name** properties in the **application.properties** file of your project.

Procedure

1. Review the **GreetingResource.java** file in your project and ensure that it contains the contents that are shown in the following example. To use the **@ConfigPoperties** annotation to inject configuration properties from another configuration source into this class, you must import the **java.util.Optional** and **org.eclipse.microprofile.config.inject.ConfigProperty** packages.

   Example GreetingResource.java file

```
import java.util.Optional;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import org.eclipse.microprofile.config.inject.ConfigProperty;

@Path("/greeting")
public class GreetingResource {

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

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

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

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

2. Create a **GreetingConfiguration.java** file in the **src/main/java/org/acme/config** directory. Add the import statements for **ConfigMapping** and **Optional** to the file:

Example GreetingConfiguration.java file

```
import io.smallrye.config.ConfigMapping;
import io.smallrye.config.WithDefault;
import java.util.Optional;

@ConfigMapping(prefix = "greeting") (1)
public interface GreetingConfiguration {
    String message();

    @WithDefault("!") (2)
    String suffix();

    Optional<String> name();
}
```

(1) The **prefix** property is optional. For example, in this scenario, the prefix is **greeting**.

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

3. Inject the **GreetingConfiguration** instance into the **GreetingResource** class by using the **@Inject** annotation, as follows:

NOTE

This snippet replaces the three fields that are annotated with **@ConfigProperty** that are in the initial version of the **config-quickstart** project.

Example GreetingResource.java file

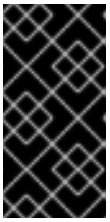
```
@Path("/greeting")
public class GreetingResource {

    @Inject
    GreetingConfiguration config;
```

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

4. Compile and start your application in development mode:

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



IMPORTANT

If you do not provide values for the class properties, the application fails to compile, and an **io.smallrye.config.ConfigValidationException** error is returned to indicate that a value is missing. This does not apply to optional fields or fields with a default value.

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

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

6. You receive the following message:

```
hello quarkus!
```

7. To stop the application, press Ctrl+C.

1.6.2. Using nested object configuration

You can define an interface that is nested inside another interface. This procedure shows how to create and configure a nested interface in the Quarkus **config-quickstart** project.

Prerequisites

- You have created the Quarkus **config-quickstart** project.
- You have defined the **greeting.message** and **greeting.name** properties in the **application.properties** file of your project.

Procedure

1. Review the **GreetingResource.java** in your project. The file contains the **GreetingResource** class with the **hello()** method that returns a message when you send an HTTP request on the /**greeting** endpoint:

   Example GreetingResource.java file

```
import java.util.Optional;

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

```
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import org.eclipse.microprofile.config.inject.ConfigProperty;

@Path("/greeting")
public class GreetingResource {

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

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

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

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

2. Create a **GreetingConfiguration.java** class file with the **GreetingConfiguration** instance. This class contains the externalized configuration for the **hello()** method that is defined in the **GreetingResource** class:

Example GreetingConfiguration.java file

```
import io.smallrye.config.ConfigMapping;
import io.smallrye.config.WithDefault;
import java.util.Optional;

@ConfigMapping(prefix = "greeting")
public interface GreetingConfiguration {
    String message();

    @WithDefault("!")
    String suffix();

    Optional<String> name();
}
```

3. Create the **ContentConfig** class that is nested inside the **GreetingConfiguration** instance, as shown in the following example:

Example GreetingConfiguration.java file

```
import io.smallrye.config.ConfigMapping;
import io.smallrye.config.WithDefault;

import java.util.List;
import java.util.Optional;
```

```
@ConfigMapping(prefix = "greeting")
public interface GreetingConfiguration {
    String message();

    @WithDefault("!")
    String suffix();

    Optional<String> name();

    ContentConfig content();

    interface ContentConfig {
        Integer prizeAmount();

        List<String> recipients();
    }
}
```

NOTE

The method name of the **ContentConfig** class is **content**. To ensure that you bind the properties to the correct interface, when you define configuration properties for this class, use **content** in the prefix. In doing so, you can also prevent property name conflicts and unexpected application behavior.

4. Define the **greeting.content.prize-amount** and **greeting.content.recipients** configuration properties in your **application.properties** file.
   The following example shows the properties defined for the **GreetingConfiguration** instance and the **ContentConfig** classes:

   Example application.properties file

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

5. Instead of the three **@ConfigProperty** field annotations, inject the **GreetingConfiguration** instance into the **GreetingResource** class by using the **@Inject** annotation, as outlined in the following example. Also, you must update the message string that the /**greeting** endpoint returns with the values that you set for the new **greeting.content.prize-amount** and **greeting.content.recipients** properties that you added.

   Example GreetingResource.java file

```
import java.util.Optional;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import jakarta.inject.Inject;
```

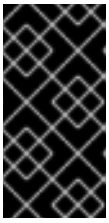
```
@Path("/greeting")
public class GreetingResource {

    @Inject
    GreetingConfiguration config;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return config.message() + " " + config.name().orElse("world") + config.suffix() + "\n" +
config.content().recipients() + " receive total of candies: " + config.content().prizeAmount();
    }
}
```

6. Compile and start your application in development mode:

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



IMPORTANT

If you do not provide values for the class properties, the application fails to compile and you receive a **jakarta.enterprise.inject.spi.DeploymentException** exception that indicates a missing value. This does not apply to **Optional** fields and fields with a default value.

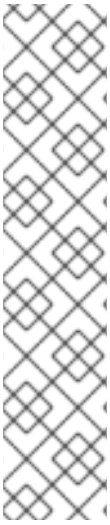
7. To verify that the endpoint returns the message, open a new terminal window and enter the following command:

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

A message displays, containing two lines of output. The first line displays the greeting, and the second line reports the recipients of the prize together with the prize amount, as follows:

```
hello quarkus!
Jane,John receive total of candies: 10
```

8. To stop the application, press Ctrl+C.



NOTE

You can annotate classes that are annotated with **@ConfigMapping** with Bean Validation annotations similar to the following example:

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

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

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

1.7. ACCESSING THE CONFIGURATION PROGRAMMATICALLY

You can define a method in your code to retrieve the values of the configuration properties in your application. In doing so, you can dynamically look up configuration property values or retrieve configuration property values from classes that are either CDI beans or Jakarta REST (formerly known as JAX-RS) resources.

You can access the configuration by 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

- Use a method to access the value of a configuration property of any class or object in your application code. Depending on whether or not the value that you want to retrieve is set in a configuration source in your project, you can use one of the following methods:
  - To access the value of a property that is set in a configuration source in your project, for example, in the **application.properties** file, use the **getValue()** method:

    ```
    String <variable-name> = ConfigProvider.getConfig().getValue("<property-name>",
    <data-type-class-name>.class);
    ```

    For example, to retrieve the value of the **greeting.message** property that has the data type **String**, and is assigned to the **message** variable in your code, use the following syntax:

    ```
    String message = config.getValue("greeting.message", String.class);
    ```

  - When you want to retrieve a value that is optional or default and might not be defined in your **application.properties** file or another configuration source in your application, use the **getOptionalValue()** method:

    ```
    String <variable-name> = ConfigProvider.getConfig().getOptionalValue("<property-
    name>", <data-type-class-name>.class);
    ```

    For example, to retrieve the value of the **greeting.name** property that is optional, has the data type **String**, and is assigned to the **name** variable in your code, use the following syntax:

    ```
    Optional<String> name = config.getOptionalValue("greeting.name", String.class);
    ```

The following snippet shows a variant of the aforementioned **GreetingResource** class by using the programmatic access to the configuration:

src/main/java/org/acme/config/GreetingResource.java

```
package org.acme.config;

import java.util.Optional;
```

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import org.eclipse.microprofile.config.Config;
import org.eclipse.microprofile.config.ConfigProvider;
import org.eclipse.microprofile.config.inject.ConfigProperty;

@Path("/greeting")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        Config config = ConfigProvider.getConfig();
        String message = config.getValue("greeting.message", String.class);
        String suffix = config.getOptionalValue("greeting.suffix", String.class).orElse("!");
        Optional<String> name = config.getOptionalValue("greeting.name", String.class);


        return message + " " + name.orElse("world") + suffix;
    }
}
```

1.8. PROPERTY EXPRESSIONS

Property expressions are combinations of property references and plain text strings that you can use to substitute values of properties in your configuration.

Much like a variable, you can use a property expression in Quarkus to substitute a value of a configuration property instead of hardcoding it. A property expression is resolved when **java.util.Properties** reads the value of the property from a configuration source in your application.

This means that if a configuration property is read from your configuration at compile time, the property expression is also resolved at compile time. If the configuration property is overridden at runtime, its value is resolved at runtime.

You can resolve property expressions by using more than one configuration source. This means that you can use a value of a property that is defined in one configuration source to expand a property expression that you use in another configuration source.

If the value of a property in an expression cannot be resolved, and you do not set a default value for the expression, your application encounters a **NoSuchElementException**.

1.8.1. Example usage of property expressions

To achieve flexibility when you configure your Quarkus application, you can use property expressions as shown in the following examples.

- Substituting the value of a configuration property:
  To avoid hardcoding property values in your configuration, you can use a property expression. Use the **${<property_name>}** syntax to write an expression that references a configuration property, as shown in the following example:

Example application.properties file

```
remote.host=quarkus.io
callable.url=https://${remote.host}/
```

The value of the **callable.url** property resolves to **https://quarkus.io/**.

- Setting a property value that is specific to a particular configuration profile:
  In the following example, the **%dev** configuration profile and the default configuration profile are set to use data source connection URLs with different host addresses.

  Example application.properties file

  ```
  %dev.quarkus.datasource.jdbc.url=jdbc:mysql://localhost:3306/mydatabase?useSSL=false
  quarkus.datasource.jdbc.url=jdbc:mysql://remotehost:3306/mydatabase?useSSL=false
  ```

  Depending on the configuration profile used to start your application, your data source driver uses the database URL that you set for the profile.

  You can achieve the same result in a simplified way by setting a different value for the custom **application.server** property for each configuration profile. Then, you can reference the property in the database connection URL of your application, as shown in the following example:

  Example application.properties file

  ```
  %dev.application.server=localhost
  application.server=remotehost

  quarkus.datasource.jdbc.url=jdbc:mysql://${application.server}:3306/mydatabase?
  useSSL=false
  ```

  The **application.server** property resolves to the appropriate value depending on the profile that you choose when you run your application.

- Setting a default value of a property expression:
  You can define a default value for a property expression. Quarkus uses the default value if the value of the property that is required to expand the expression is not resolved from any of your configuration sources. You can set a default value for an expression by using the following syntax:

  ```
  ${<property_name>:<default_value>}
  ```

  In the following example, the property expression in the data source URL uses **mysql.db.server** as the default value of the **application.server** property:

  Example application.properties file

  ```
  quarkus.datasource.jdbc.url=jdbc:mysql://${application.server:mysql.db.server}:3306/mydatabas
  e?useSSL=false
  ```

- Nesting property expressions:

You can compose property expressions by nesting a property expression inside another property expression. When nested property expressions are expanded, the inner expression is expanded first. You can use the following syntax for nesting property expressions:

```
${<outer_property_name>${<inner_property_name>}}
```

- Combining multiple property expressions:
  You can join two or more property expressions together by using the following syntax:

```
${<first_property_name>}${<second_property_name>}
```

- Combining property expressions with environment variables:
  You can use property expressions to substitute the values of environment variables. The expression in the following example substitutes the value that is set for the **HOST** environment variable as the value of the **application.host** property:

  Example application.properties file

```
remote.host=quarkus.io
application.host=${HOST:${remote.host}}
```

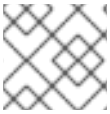
When the **HOST** environment variable is not set, the **application.host** property uses the value of the **remote.host** property as the default.

1.9. 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 to select between them by using a profile name.

Red Hat build of Quarkus has the following three default configuration profiles:

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



NOTE

In addition, you can create your own custom profiles.

Prerequisites

You have a Quarkus Maven project.

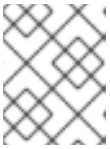
Procedure

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 by invoking the **ProfileManager.getActiveProfile()** method:

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



NOTE

You cannot access the current profile by using the **@ConfigProperty("quarkus.profile")** method.

1.9.1. 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 a configuration by 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 **<property_name>** is the name of the property, **<value>** is the property value, and **<profile>** is the name of a profile:

Create a configuration property

```
%<profile>.<property_name>=<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:

Example configuration

```
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:

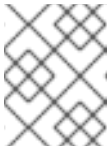
Enable a profile using quarkus.profile property

```
mvn -Dquarkus.profile=<value> quarkus:dev
```

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

Enable a profile using an environment variable

```
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:

Change a profile

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

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

Example command to activate a 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 that is 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 as a JAR, Quarkus is in **prod** mode.

1.10. SETTING CUSTOM CONFIGURATION SOURCES

By default, a Quarkus application reads properties from the **application.properties** file in the **src/main/resources** subdirectory of your project. Quarkus also allows you to load application configuration properties from other sources according to the MicroProfile Config specification for externalized configuration. You can enable your application to load configuration properties from other sources by defining 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 in your Quarkus project.

Prerequisite

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

Procedure

1. Create a class file in your project that implements the **org.eclipse.microprofile.config.spi.ConfigSourceProvider** interface. To return a list of **ConfigSource** objects, you must override the **getConfigSources()** method.

Example org.acme.config.InMemoryConfigSourceProvider

```
package org.acme.config;

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

import java.util.List;
```

```
public class InMemoryConfigSourceProvider implements ConfigSourceProvider {

    @Override
    public Iterable<ConfigSource> getConfigSources(ClassLoader classLoader) {
    return List.of(new InMemoryConfigSource());
    }
}
```

2. Create the **InMemoryConfigSource** class that implements the **org.eclipse.microprofile.config.spi.ConfigSource** interface:

Example org.acme.config.InMemoryConfigSource

```
package org.acme.config;

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

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

public class InMemoryConfigSource implements ConfigSource {
private static final Map<String, String> configuration = new HashMap<>();


  static {
     configuration.put("my.prop", "1234");
  }

   @Override
   public int getOrdinal() { (1)
     return 275;
   }

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

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

   @Override
   public String getName() {
     return InMemoryConfigSource.class.getSimpleName();
   }
}
```

(1) The **getOrdinal()** method returns the priority of the **ConfigSource** class. Therefore, when multiple configuration sources define the same property, Quarkus can select the appropriate value as defined by the **ConfigSource** class with the highest priority.

3. In the **src/main/resources/META-INF/services/** subdirectory of your project, create a file named **org.eclipse.microprofile.config.spi.ConfigSourceProvider** and enter the fully-qualified name of the class that implements the **ConfigSourceProvider** in the file that you created:

   Example org.eclipse.microprofile.config.spi.ConfigSourceProvider file:

   ```
   org.acme.config.InMemoryConfigSourceProvider
   ```

   To ensure that the **ConfigSourceProvider** that you created is registered and installed when you compile and start your application, you must complete the previous step.

4. Edit the **GreetingResource.java** file in your project to add the following update:

   ```
   @ConfigProperty(name="my.prop") int value;
   ```

5. In the **GreetingResource.java** file, extend the **hello** method to use the new property:

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

6. To compile and start your application in development mode, enter the following command:

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

7. To verify that the /**greeting** endpoint returns the expected message, open a terminal window and enter the following command:

   Example request

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

8. When your application successfully reads the custom configuration, the command returns the following response:

   ```
   hello world 1234
   ```

1.11. 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. By using converters, you can transform the string representation of a value into an object.

Prerequisites

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

Procedure

1. In the **org.acme.config package**, create the **org.acme.config.MyCustomValue** class with the following content:

   Example of custom configuration value

```
package org.acme.config;

public class MyCustomValue {

  private final int value;

  public MyCustomValue(Integer value) {
    this.value = value;
  }

  public int value() {
    return value;
  }
}
```

2. Implement the converter class to override the convert method to produce a **MyCustomValue** instance.

   Example implementation of converter class

```
package org.acme.config;

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

public class MyCustomValueConverter implements Converter<MyCustomValue> {

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

3. 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:

   Example org.eclipse.microprofile.config.spi.Converter file

```
org.acme.config.MyCustomValueConverter
org.acme.config.SomeOtherConverter
org.acme.config.YetAnotherConverter
```

4. In the **GreetingResource.java** file, inject the **MyCustomValue** property:

```
@ConfigProperty(name="custom")
    MyCustomValue value;
```

5. Edit the **hello** method to use this value:

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

6. In the **application.properties** file, add the string representation to be converted:

```
custom=1234
```

7. To compile and start your application in development mode, enter the following command:

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

8. To verify that the /**greeting** endpoint returns the expected message, open a terminal window and enter the following command:

   Example request

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

9. When your application successfully reads the custom configuration, the command returns the following response:

```
hello world - 1234
```

NOTE

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

Additional resources:

- List of converters in the **microprofile-config** GitHub repository

1.11.1. Setting custom converters priority

The default priority for all Quarkus core converters is 200. For all other converters, the default priority is 100. You can increase the priority of your custom converters by using the **jakarta.annotation.Priority** annotation.

The following procedure demonstrates an implementation of a custom converter, **AnotherCustomValueConverter**, which has a priority of 150. This takes precedence over **MyCustomValueConverter** from the previous section, which has a default priority of 100.

Prerequisites

- You have created the Quarkus **config-quickstart** project.
- You have created a custom configuration converter for your application.

Procedure

1. Set a priority for your custom converter by annotating the class with the **@Priority** annotation and passing it a priority value. In the following example, the priority value is set to **150**.

   Example AnotherCustomValueConverter.java file

```
package org.acme.config;

import jakarta.annotation.Priority;
import org.eclipse.microprofile.config.spi.Converter;

@Priority(150)
public class AnotherCustomValueConverter implements Converter<MyCustomValue> {

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

2. Create a file named **org.eclipse.microprofile.config.spi.Converter** in the **src/main/resources/META-INF/services/** subdirectory of your project, and enter the fully qualified name of the class that implements the **Converter** in the file that you created:

   Example org.eclipse.microprofile.config.spi.Converter file

```
org.acme.config.AnotherCustomValueConverter
```

   You must complete the previous step to ensure that the **Converter** you created is registered and installed when you compile and start your application.

Verification

After you complete the required configuration, the next step is to compile and package your Quarkus application. For more information and examples, see the compiling and packaging sections of the Getting started with Quarkus guide.

1.12. ADDITIONAL RESOURCES

- Developing and compiling your Quarkus applications with Apache Maven
- Deploying your Quarkus applications to OpenShift Container Platform
- Compiling your Quarkus applications to native executables

Revised on 2024-04-04 11:41:53 UTC