Red Hat build of Quarkus 1.11

Configuring your Quarkus applications

Last Updated: 2021-06-15

Legal Notice

Copyright © 2021 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide describes how to configure Quarkus applications.

Table of Contents

PREFACE 3

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION 4

MAKING OPEN SOURCE MORE INCLUSIVE 5

CHAPTER 1. RED HAT BUILD OF QUARKUS CONFIGURATION OPTIONS 6

CHAPTER 2. CREATING THE CONFIGURATION QUICKSTART PROJECT 7

CHAPTER 3. GENERATING AN EXAMPLE CONFIGURATION FILE FOR YOUR QUARKUS APPLICATION 8

CHAPTER 4. INJECTING CONFIGURATION VALUES INTO YOUR QUARKUS APPLICATION 9
- 4.1. ANNOTATING A CLASS WITH @CONFIGPROPERTIES 10
- 4.2. USING NESTED OBJECT CONFIGURATION 13
- 4.3. ANNOTATING AN INTERFACE WITH @CONFIGPROPERTIES 15

CHAPTER 5. ACCESSING THE CONFIGURATION FROM CODE 18

CHAPTER 6. SETTING CONFIGURATION PROPERTIES 19

CHAPTER 7. PROPERTY EXPRESSIONS 21
- 7.1. EXAMPLE USAGE OF PROPERTY EXPRESSIONS 21

CHAPTER 8. USING CONFIGURATION PROFILES 23
- 8.1. SETTING A CUSTOM CONFIGURATION PROFILE 23

CHAPTER 9. SETTING CUSTOM CONFIGURATION SOURCES 25

CHAPTER 10. USING CUSTOM CONFIGURATION CONVERTERS AS CONFIGURATION VALUES 28
- 10.1. SETTING CUSTOM CONVERTERS PRIORITY 28

CHAPTER 11. ADDING YAML CONFIGURATION SUPPORT 30
- 11.1. USING NESTED OBJECT CONFIGURATION WITH YAML 30
- 11.2. SETTING CUSTOM CONFIGURATION PROFILES WITH YAML 31
- 11.3. MANAGING CONFIGURATION KEY CONFLICTS 32

CHAPTER 12. UPDATING THE FUNCTIONAL TEST TO VALIDATE CONFIGURATION CHANGES 33

CHAPTER 13. PACKAGING AND RUNNING YOUR QUARKUS APPLICATION 34

CHAPTER 14. ADDITIONAL RESOURCES 35

PREFACE

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.

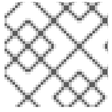
This guide describes how to configure a Quarkus application using the Eclipse MicroProfile Config method or YAML format. The procedures include configuration examples created using the Quarkus **config-quickstart** exercise.

Prerequisites

- Have OpenJDK (JDK) 11 installed and the **JAVA_HOME** environment variable specifies the location of the Java SDK.
  - Log in the Red Hat Customer Portal to download the Red Hat build of Open JDK from the Software Downloads page.
- Have Apache Maven 3.8.1 or higher installed.
  - Download Maven from the Apache Maven Project website.
- Have Maven configured to use artifacts from the Quarkus Maven repository.
  - To learn how to configure Maven settings see Getting started with Quarkus.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our technical content and encourage you to tell us what you think. If you’d like to add comments, provide insights, correct a typo, or even ask a question, you can do so directly in the documentation.



NOTE

You must have a Red Hat account and be logged in to the customer portal.

To submit documentation feedback from the customer portal, do the following:

1. Select the **Multi-page HTML** format.
2. Click the **Feedback** button at the top-right of the document.
3. Highlight the section of text where you want to provide feedback.
4. Click the **Add Feedback** dialog next to your highlighted text.
5. Enter your feedback in the text box on the right of the page and then click **Submit**.

We automatically create a tracking issue each time you submit feedback. Open the link that is displayed after you click **Submit** and start watching the issue or add more comments.

Thank you for the valuable feedback.

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. 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. Verify that Maven is using JDK 11 and that the Maven version is 3.8.1 or higher:

   ```
   mvn --version
   ```

   If this command does not return JDK 11, ensure that the directory where JDK 11 is installed on your system is included in the **PATH** environment variable:

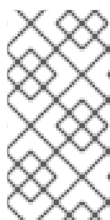
   ```
   export PATH=$PATH:/path/to/jdk-11
   ```

2. Enter the following command to generate the project:

   ```
   mvn io.quarkus:quarkus-maven-plugin:1.11.7.Final-redhat-00009:create \
       -DprojectGroupId=org.acme \
       -DprojectArtifactId=config-quickstart \
       -DplatformGroupId=com.redhat.quarkus \
       -DplatformVersion=1.11.7.Final-redhat-00009 \
       -DclassName="org.acme.config.GreetingResource" \
       -Dpath="/greeting"
   cd config-quickstart
   ```

   This 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**, **Dockerfile.native** and **Dockerfile.fast-jar** files in **src/main/docker**
   - 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 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 that 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

- 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
  ```

Additional resources

- Quarkus Community Documentation - 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

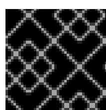
1. Open the **src/main/resources/application.properties** file.
2. Add configuration properties to your configuration file where **<key>** is the property name and **<value>** is the value of the property:

```
<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:

src/main/resources/application.properties

```
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 corresponding properties by annotating them with **@ConfigProperty** as shown in the following example:

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

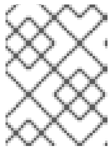
```
@ConfigProperty(name = "greeting.message") (1)
String message;

@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: **javax.enterprise.inject.spi.DeploymentException: No config value of type [class java.lang.String] exists for: greeting.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:

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

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

6. Compile and start your application in development mode:

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

7. 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!
   ```

8. Press **CTRL+C** to stop the application.

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:

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

   ```
   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:

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

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

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

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

   ```
   @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 specify a prefix, it is determined by the name of the class. In this example, the prefix is **greeting**.
2. If **greeting.suffix** is not set, **!** is the default value.

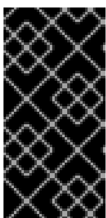
5. Inject the **GreetingConfiguration** class into the **GreetingResource** class using the **@Inject** annotation:

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

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

    @Inject
    GreetingConfiguration config;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return config.getMessage() + " " + config.getName().orElse("world") + config.getSuffix();
    }
}
```

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 **javax.enterprise.inject.spi.DeploymentException** that indicates a missing value. This does not apply to **Optional** fields and fields with a default value.

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

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

8. You receive the following message:

```
hello quarkus!
```

9. Press **CTRL+C** to stop the application.

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:

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

   ```
   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:

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

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

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

3. Add a nested class inside the **GreetingConfiguration** class as shown in the following example:

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

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

       public String message;
       public String suffix = "!";
       public Optional<String> name;
       public ContentConfig content;

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

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

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

   src/main/resources/application.properties

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

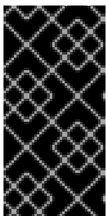
5. Inject the **GreetingConfiguration** class into the **GreetingResource** class using the **@Inject** annotation, and update the message string that the /**greeting** endpoint returns to have the message show the values that you set for the new **greeting.content.prize-amount** and **greeting.content.recipients** properties that you added:

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

   ```
   @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 **javax.enterprise.inject.spi.DeploymentException** that indicates a missing value. This does not apply to **Optional** fields and fields with a default value.

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

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

8. You receive the message that contains the greeting on the first line and the recipients of the prize and the prize amount on the second line:

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

9. Press **CTRL+C** to stop the application.

> **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:

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

   ```
   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:

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

   ```
   @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.

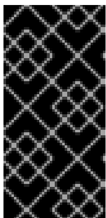
3. Inject the **GreetingConfiguration** class into the **GreetingResource** class using the **@Inject** annotation:

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

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

    @Inject
    GreetingConfiguration config;

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

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 you receive a **javax.enterprise.inject.spi.DeploymentException** that indicates a missing value. This does not apply to **Optional** fields and fields with a default value.

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

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

6. You receive the following message:

```
hello quarkus!
```

7. Press **CTRL+C** to stop the application.

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 either 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:

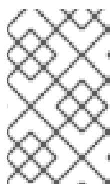
```
java -D<key>=<value> -jar target/myapp-runner.jar
```

     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
```

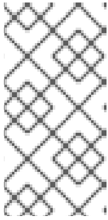


NOTE

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 **<PROPERTY_KEY>** is the property name and **<value>** is the value of the property:

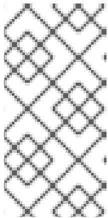
```
<PROPERTY_KEY>=<value>
```



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. 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 form your configuration at compile time, the property expression is also resolved at compile time. If the configuration property is overriden at runtime, its value is resolved at runtime.

Property expressions can be resolved 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**.

7.1. EXAMPLE USAGE OF PROPERTY EXPRESSIONS

In this section you can find examples of how you can use property expressions to achieve greater flexibility when configuring of your Quarkus application.

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

  application.properties

  ```
  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. Depending on the configuration profile with which you start your application, your data source driver uses the database URL that you set for the profile:

  application.properties

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

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

  application.properties

```
%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 of an expression using the following syntax:

  ```
  ${<expression>:<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:

  application.properties

  ```
  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:

  ```
  ${<outer_property_expression>${<inner_property_expression>}}
  ```

- Multiple property expressions:
  You can join two or more property expression together as shown below:

  ```
  ${<first_property>}${<second_property>}
  ```

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

  application.properties

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

CHAPTER 8. 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.

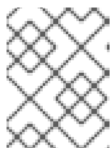
Procedure

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.

8.1. 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.

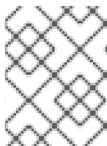
     - To enable a profile using the **quarkus.profile** system property, enter the following command:

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

   - Set the **QUARKUS_PROFILE** environment variable.

     - To enable profile using an environment variable, enter the following command:

       ```
       export QUARKUS_PROFILE=<profile>
       ```

       NOTE

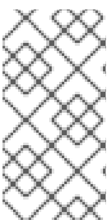
       The system property value takes precedence over the environment variable value.

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 as a JAR, Quarkus is in **prod** mode.

CHAPTER 9. SETTING CUSTOM CONFIGURATION SOURCES

By default, your Quarkus application reads properties from the **application.properties** file in the **src/main/resources** subdirectory of your project. However, because Quarkus supports MicroProfile Config, you can also load the configuration of your application from other sources. 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 in your Quarkus project.

Prerequisite

- Have the Quarkus **config-quickstart** project.

Procedure

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

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

```
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 that implements the **org.eclipse.microprofile.config.spi.ConfigSourceProvider** interface. You must override its **getConfigSources** method to return a list of **ConfigSource** objects:
   The following example shows a custom implementation of the **ConfigSourceProvider** and the **ConfigSource** interfaces:

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

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

   src/main/resources/META-INF/services/org.eclipse.microprofile.config.spi.ConfigSourceProvider

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

   You must perform this step to ensure that the **ConfigSourceProvider** that you created is registered and installed when you compile and start your application.

4. Enter the following command to compile and start your application in development mode:

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

5. Enter the following command in a new terminal window to verify that the **/greeting** endpoint returns the expected message:

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

6. When your application reads the custom configuration properly, you receive the following response.

```
hello quarkus!
```

CHAPTER 10. 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.

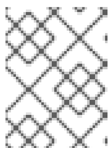
Procedure

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:

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

10.1. 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.

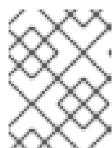
Procedure

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 11. 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"
    ```

11.1. USING NESTED OBJECT CONFIGURATION WITH YAML

You can define a nested class inside an already existing class. This procedure shows how you can set nested configuration properties for your Quarkus application using a configuration file in YAML format.

Prerequisites

- Have a Quarkus Maven project.
- Have a PostgreSQL data source.
- Have the following extensions as dependencies in the **pom.xml** file of your project:
  - **quarkus-rest-client**,
  - **quarkus-jdbc-postgresql**
  - **quarkus-config-yaml**

Procedure

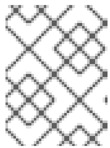
1. Open the **src/main/resources/application.yaml** configuration file.
2. Add the nested class configuration properties to your **application.yaml** file as shown in the example:

   src/main/resources/application.yaml

   _

```
# Properties that configure the JDBC data source driver of your PostgreSQL data source
quarkus:
  datasource:
    url: jdbc:postgresql://localhost:5432/some-database
    driver: org.postgresql.Driver
    username: quarkus
    password: quarkus

# Property that configures the URL of the endpoint to which the rest client sends requests
org:
  acme:
    restclient:
      CountriesService/mp-rest/url: https://restcountries.eu/rest

# Property that configures the log message level for your application
quarkus:
  log:
    category:
      # Do not use spaces in names of configuration properties that you place inside quotation
marks
      "io.quarkus.category":
        level: INFO
```

Note, that you can use comments to describe your configuration properties in a similar way as you use them in **application.properties**.



NOTE

Always use spaces to indent the properties in your YAML configuration file. YAML does not allow using tabs for indentation.

11.2. SETTING CUSTOM CONFIGURATION PROFILES WITH YAML

With Quarkus you can set configuration properties and values that are specific to different configuration profiles of your application. You can start your application with a specific profile to access a particular configuration. This procedure demonstrates how you can provide a configuration for a specific profile in YAML format.

Prerequisites

- Have a Quarkus Maven project that is configured to use a PostgreSQL data source with a JDBC data source driver.
- Have the **quarkus-jdbc-postgresql** and **quarkus-config-yaml** extensions as dependencies in the **pom.xml** file of your project.

Procedure

1. Open the **src/main/resources/application.yaml** configuration file.
2. To set a profile dependent configuration, add the profile name before defining the key-value pairs using the **"%<profile_name>"** syntax. Ensure that you place the profile name inside quotation marks. In YAML, all strings that begin with a special character must be placed inside quotation marks.

In the following example the PostgreSQL database is configured to be available at the **jdbc:postgresql://localhost:5432/some-database** URL when you start your Quarkus application in development mode:

src/main/resources/application.yaml

```
"%dev":
  # Properties that configure the JDBC data source driver of your PostgreSQL data source
  quarkus:
    datasource:
      url: jdbc:postgresql://localhost:5432/some-database
      driver: org.postgresql.Driver
      username: quarkus
      password: quarkus
```

11.3. 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.

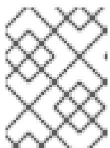
Procedure

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 12. 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 13. 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
   ```

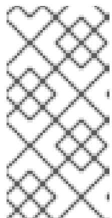
   This command produces the following JAR files in the /**target** directory:

   - **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.

CHAPTER 14. ADDITIONAL RESOURCES

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

Revised on 2021-06-15 14:50:23 UTC