Quarkus - Configuring Your Application

Hardcoded values in your code are a *no go* (even if we all did it at some point ;-)). In this guide, we learn how to configure your application.

Prerequisites

To complete this guide, you need:

- between 5 and 10 minutes

- an IDE

- JDK 1.8+ installed with JAVA_HOME configured appropriately

- Apache Maven 3.6.3

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: git clone https://github.com/quarkusio/quarkus-quickstarts.git, or download an archive.

The solution is located in the config-quickstart directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=config-quickstart \
    -DclassName="org.acme.config.GreetingResource" \
    -Dpath="/greeting"
cd config-quickstart
```

It generates:

- the Maven structure

- a landing page accessible on http://localhost:8080

- example Dockerfile files for both native and jvm modes

- the application configuration file

- an `org.acme.config.GreetingResource` resource

- an associated test

Injecting configuration value

Quarkus uses MicroProfile Config to inject the configuration in the application. The injection uses the `@ConfigProperty` annotation.

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

> **ℹ** When injecting a configured value, you can use `@Inject @ConfigProperty` or just `@ConfigProperty`. The `@Inject` annotation is not necessary for members annotated with `@ConfigProperty`, a behavior which differs from MicroProfile Config

Edit the `org.acme.config.GreetingResource`, and introduce the following configuration properties:

```
@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 startup fails with `javax.enterprise.inject.spi.DeploymentException: No config value of type [class java.lang.String] exists for: greeting.message`.

② The default value is injected if the configuration does not provide a value for `greeting.suffix`.

③ This property is optional - an empty `Optional` is injected if the configuration does not provide a value for `greeting.name`.

Now, modify the `hello` method to use the injected properties:

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

Create the configuration

By default, Quarkus reads `application.properties`. Edit the `src/main/resources/application.properties` with the following content:

```
# Your configuration properties
greeting.message = hello
greeting.name = quarkus
```

Once set, check the application with:

```
$ curl http://localhost:8080/greeting
hello quarkus!
```

> 💡 If the application requires configuration values and these values are not set, an error is thrown. So you can quickly know when your configuration is complete.

Update the test

We also need to update the functional test to reflect the changes made to the endpoint. Edit the `src/test/java/org/acme/config/GreetingResourceTest.java` file and change the content of the `testHelloEndpoint` method to:

```java
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
    }

}
```

Package and run the application

Run the application with: `./mvnw compile quarkus:dev`. Open your browser to http://localhost:8080/greeting.

Changing the configuration file is immediately reflected. You can add the `greeting.suffix`, remove the other properties, change the values, etc.

As usual, the application can be packaged using `./mvnw clean package` and executed using the `-runner.jar` file. You can also generate the native executable with `./mvnw clean package -Pnative`.

Programmatically access the configuration

You can access the configuration programmatically. It can be handy to achieve dynamic lookup, or retrieve configured values from classes that are neither CDI beans or JAX-RS resources.

You can access the configuration programmatically using `org.eclipse.microprofile.config.ConfigProvider.getConfig()` such as in:

```
String databaseName =
ConfigProvider.getConfig().getValue("database.name", String.class);
Optional<String> maybeDatabaseName =
ConfigProvider.getConfig().getOptionalValue("database.name",
String.class);
```

Using @ConfigProperties

As an alternative to injecting multiple related configuration values in the way that was shown in the previous example, users can also use the `@io.quarkus.arc.config.ConfigProperties` annotation to group these properties together.

For the greeting properties above, a `GreetingConfiguration` class could be created like so:

```
package org.acme.config;

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

@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 not set then the prefix to be used will be determined by the class name. In this case it would still be `greeting` (since the `Configuration` suffix is removed). If the class were named `GreetingExtraConfiguration` then the resulting default prefix would be `greeting-extra`.

② `!` will be the default value if `greeting.suffix` is not set

This class could then be injected into the `GreetingResource` using the familiar CDI `@Inject` annotation like so:

```
@Inject
GreetingConfiguration greetingConfiguration;
```

Another alternative style provided by Quarkus is to create `GreetingConfiguration` as an interface like so:

```
package org.acme.config;

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

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

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

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

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

① The `@ConfigProperty` annotation is needed because the name of the configuration property that the method corresponds to doesn't follow the getter method naming conventions

② In this case since `name` was not set, the corresponding property will be `greeting.suffix`.

③ It is unnecessary 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.

When using `@ConfigProperties` on a class or an interface, if the value of one of its fields is not provided, the application startup will fail and a `javax.enterprise.inject.spi.DeploymentException` indicating the missing value information will be thrown. This does not apply to `Optional` fields and fields with a default value.

Additional notes on @ConfigProperties

When using a regular class annotated with `@ConfigProperties` the class doesn't necessarily have to declare getters and setters. Having simple public non-final fields is valid as well.

Furthermore, the configuration classes support nested object configuration. Suppose there was a need to have an extra layer of greeting configuration named `hidden` that would contain a few fields. This could be achieved like so:

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

    public String message;
    public String suffix = "!";
    public Optional<String> name;
    public HiddenConfig hidden; ①

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

① The name of the field (not the class name) will determine the name of the properties that are bound to the object.

Setting the properties would occur in the normal manner, for example in `application.properties` one could have:

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

Furthermore, 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 = "!";

}
```

> ⚠️ For validation to work, the `quarkus-hibernate-validator` extension needs to be present.

If the validation fails with the given configuration, the application will fail to start and indicate the corresponding validation errors in the log.

In the case of an interface being annotated with `@ConfigProperties`, the interface is allowed to extend other interfaces and methods from the entire interface hierarchy are used to bind properties.

Using same ConfigProperties with different prefixes

Quarkus also supports the use of the same `@ConfigProperties` object with different prefixes for each injection point using the `io.quarkus.arc.config.@ConfigPrefix` annotation. Say for example that `GreetingConfiguration` from above needs to be used for both the `greeting` prefix and the `other` prefix. In that case the code would look like so:

`GreetingConfiguration.java`

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

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

}
```

`SomeBean.java`

```java
@ApplicationScoped
public class SomeBean {

    @Inject ①
    GreetingConfiguration greetingConfiguration;

    @ConfigPrefix("other") ②
    GreetingConfiguration otherConfiguration;

}
```

① At this injection point `greetingConfiguration` will use the `greeting` prefix since that is what has been defined on `@ConfigProperties`.

② At this injection point `otherConfiguration` will use the `other` prefix from `@ConfigPrefix` instead of the `greeting` prefix. Notice that in this case `@Inject` is not required.

Configuring Quarkus

Quarkus itself is configured via the same mechanism as your application. Quarkus reserves the `quarkus.` namespace for its own configuration. For example to configure the HTTP server port you can set `quarkus.http.port` in `application.properties`.

As mentioned above, properties prefixed with `quarkus.` are effectively reserved for configuring Quarkus itself and therefore `quarkus.` should **never** be used as prefix for application specific properties.

In the previous examples using `quarkus.message` instead of `greeting.message` would result in unexpected behavior.

List of all configuration properties

All the Quarkus configuration properties are documented and searcheable.

Generating configuration for your application

It is also possible to generate an example `application.properties` with all known configuration properties, to make it easy to see what Quarkus configuration options are available. To do this, run:

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

This will create a `src/main/resources/application.properties.example` file that contains all the config options exposed via the extensions you currently have installed. These options are commented out, and have their default value when applicable. For example this HTTP port config entry will appear as:

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

Rather than generating an example config file, you can also add these to you actual config file by setting the `-Dfile` parameter:

```
./mvnw quarkus:generate-config -Dfile=application.properties
```

If a config option is already present (commented or not) it will not be added, so it is safe to run this after adding an additional extension to see what additional options have been added.

Overriding properties at runtime

Quarkus does much of its configuration and bootstrap at build time. Most properties will then be read and set during the build time step. To change them, make sure to repackage your application.

```
./mvnw clean package
```

Extensions do define *some* properties as overridable at runtime. A canonical example is the database URL, username and password which is only known specifically in your target environment. This is a tradeoff as the more runtime properties are available, the less build time prework Quarkus can do. The list of runtime properties is therefore lean.

You can override these runtime properties with the following mechanisms (in decreasing priority):

1. using system properties:

   ◦ for a runner jar: `java -Dquarkus.datasource.password=youshallnotpass -jar target/myapp-runner.jar`

   ◦ for a native executable: `./target/myapp-runner -Dquarkus.datasource.password=youshallnotpass`

2. using environment variables:

   ◦ for a runner jar: `export QUARKUS_DATASOURCE_PASSWORD=youshallnotpass ; java -jar target/myapp-runner.jar`

   ◦ for a native executable: `export QUARKUS_DATASOURCE_PASSWORD=youshallnotpass ; ./target/myapp-runner`

3. using an environment file named `.env` placed in the current working directory containing the line `QUARKUS_DATASOURCE_PASSWORD=youshallnotpass` (for dev mode, this file can be placed in the root of the project, but it is advised to not check it in to version control)

4. using a configuration file placed in `$PWD/config/application.properties`

   ◦ By placing an `application.properties` file inside a directory named `config` which resides in the directory where the application runs, any runtime properties defined in that file will override the default configuration. Furthermore any runtime properties added to this file that were not part of the original `application.properties` file *will also* be taken into account.

   ◦ This works in the same way for runner jar and the native executable

   > 🛈 Environment variables names are following the conversion rules of Eclipse MicroProfile

   > 🛈 The `config/application.properties` features is available in development mode as well. To make use of it, `config/application.properties` needs to be placed inside the build tool's output directory (`target` for Maven and `build/classes/java/main` for Gradle). Keep in mind however that any cleaning operation from the build tool like `mvn clean` or `gradle clean` will remove the `config` directory as well.

Configuration Profiles

Quarkus supports the notion of configuration profiles. These allow you to have multiple configuration in the same file and select between them via a profile name.

The syntax for this is `%{profile}.config.key=value`. For example if I have the following:

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

The Quarkus HTTP port will be 9090, unless the dev profile is active, in which case it will be 8181.

By default Quarkus has three profiles, although it is possible to use as many as you like. The default profiles are:

- **dev** - Activated when in development mode (i.e. `quarkus:dev`)

- **test** - Activated when running tests

- **prod** - The default profile when not running in development or test mode

There are two ways to set a custom profile, either via the `quarkus.profile` system property or the `QUARKUS_PROFILE` environment variable. If both are set the system property takes precedence. Note that it is not necessary to define the names of these profiles anywhere, all that is necessary is to create a config property with the profile name, and then set the current profile to that name. For example if I want a `staging` profile with a different HTTP port I can add the following to `application.properties`:

```
quarkus.http.port=9090
%staging.quarkus.http.port=9999
```

And then set the `QUARKUS_PROFILE` environment variable to `staging` to activate my profile.

> ℹ️ The proper way to check the active profile programmatically is to use the `getActiveProfile` method of `io.quarkus.runtime.configuration.ProfileManager`.
>
> Using `@ConfigProperty("quarkus.profile")` will **not** work properly.

Using Property Expressions

Quarkus supports the use of property expressions in the `application.properties` file.

These expressions are resolved when the property is read. So if your configuration property is a build time configuration property, the property expression will be resolved at build time. If your configuration property is overridable at runtime, the property expression will be resolved at runtime.

You can use property expressions both for the Quarkus configuration or for your own configuration properties.

Property expressions are defined this way: `${my-property-expression}`.

For example, having the following property:

```
remote.host=quarkus.io
```

and another property defined as:

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

will result in the value of the `callable.url` property being set to:

```
callable.url=https://quarkus.io/
```

Another example would be defining different database servers depending on the profile used:

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

can be simplified by having:

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

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

It does result in one more line in this example but the value of `application.server` can be reused in other properties, diminishing the possibility of typos and providing more flexibility in property definitions.

Combining Property Expressions and Environment Variables

Quarkus also supports the combination of both property expressions and environment variables.

Let's assume you have following property defined in `application.properties`:

```
remote.host=quarkus.io
```

You can combine environment variables and property expressions by having a property defined as follows:

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

This will expand the `HOST` environment variable and use the value of the property `remote.host` as the default value if `HOST` is not set.

For the purpose of this section we used the property `remote.host` we defined previously. It has to be noted that the value could have been a fixed one such as in:

```
application.host=${HOST:localhost}
```

which will result in `localhost` as the default value if `HOST` is not set.

Clearing properties

Run time properties which are optional, and which have had values set at build time or which have a default value, may be explicitly cleared by assigning an empty string to the property. Note that this will *only* affect run time properties, and will *only* work with properties whose values are not required.

The property may be cleared by setting the corresponding `application.properties` property, setting the corresponding system property, or setting the corresponding environment variable.

Miscellaneous

The default Quarkus application runtime profile is set to the profile used to build the application. For example:

```
./mvnw package -Pnative -Dquarkus.profile=prod-aws
./target/my-app-1.0-runner ①
```

① The command will run with the `prod-aws` profile. This can be overridden using the `quarkus.profile` system property.

Custom Configuration

Custom configuration sources

You can also introduce custom configuration sources in the standard MicroProfile Config manner. To do this, you must provide a class which implements either `org.eclipse.microprofile.config.spi.ConfigSource` or `org.eclipse.microprofile.config.spi.ConfigSourceProvider`. Create a service file for the class and it will be detected and installed at application startup.

Custom configuration converters

You can also use your custom types as a configuration values. This can be done by implementing `org.eclipse.microprofile.config.spi.Converter<T>` and adding its fully qualified class name in the `META-INF/services/org.eclipse.microprofile.config.spi.Converter` file.

Let us assume you have a custom type like this one:

```
package org.acme.config;

public class MicroProfileCustomValue {

    private final int number;

    public MicroProfileCustomValue(int number) {
        this.number = number;
    };

    public int getNumber() {
        return number;
    }
}
```

The corresponding converter will look like the one below. Please note that your custom converter class must be `public` and must have a `public` no-argument constructor. It also must not be `abstract`.

```
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));
    }
}
```

Then you need to include the fully qualified class name of the converter in a service file `META-INF/services/org.eclipse.microprofile.config.spi.Converter`. If you have more converters, simply add their class names in this file as well. Single fully qualified class name per line, for example:

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

Please note that `SomeOtherConverter` and `YetAnotherConverter` were added just for a demonstration. If you include in this file classes which are not available at runtime, the converters loading will fail.

After this is done you can use your custom type as a configuration value:

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

Converter priority

In some cases, you may want to use a custom converter to convert a type which is already converted by a different converter. In such cases, you can use the `javax.annotation.Priority` annotation to change converters precedence and make your custom converter of higher priority than the other on the list.

By default, if no `@Priority` can be found on a converter, it's registered with a priority of 100 and all Quarkus core converters are registered with a priority of 200, so depending on which converter you would like to replace, you need to set a higher value.

To demonstrate the idea let us implement a custom converter which will take precedence over `MicroProfileCustomValueConverter` implemented in the previous example.

```
package org.acme.config;

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

@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);
    }
}
```

Since it converts the same value type (namely `MicroProfileCustomValue`) and has a priority of 150, it will be used instead of a `MicroProfileCustomValueConverter` which has a default priority of 100.

> ℹ️ This new converter also needs to be listed in a service file, i.e. `META-INF/services/org.eclipse.microprofile.config.spi.Converter`.

YAML for Configuration

Add YAML Config Support

You might want to use YAML over properties for configuration. Since SmallRye Config brings support for YAML configuration, Quarkus supports this as well.

First you will need to add the YAML extension to your `pom.xml`:

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

Or you can alternatively run this command in the directory containing your Quarkus project:

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

Now Quarkus can read YAML configuration files. The config directories and priorities are the same as before.

> **i** Quarkus will choose an `application.yaml` over an `application.properties`. YAML files are just an alternative way to configure your application. You should decide and keep one configuration type to avoid errors.

Configuration Examples

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

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

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

> **i** Quarkus also supports using `application.yml` as the name of the YAML file. The same rules apply for this file as for `application.yaml`.

Profile dependent configuration

Providing profile dependent configuration with YAML is done like with properties. Just add the `%profile` wrapped in quotation marks before defining the key-value pairs:

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

Configuration key conflicts

The MicroProfile Configuration specification defines configuration keys as an arbitrary `.`-delimited string. However, structured formats like YAML naively only support a subset of the possible configuration namespace. For example, consider the two configuration properties `quarkus.http.cors` and `quarkus.http.cors.methods`. One property is the prefix of another, so it may not be immediately evident how to specify both keys in your YAML configuration.

This is solved by using a null key (normally represented by `~`) for any YAML property which is a prefix of another one. Here's an example:

An example YAML configuration resolving prefix-related key name conflicts

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

In general, null YAML keys are not included in assembly of the configuration property name, allowing them to be used to any level for disambiguating configuration keys.

More info on how to configure

Quarkus relies on SmallRye Config and inherits its features.

SmallRye Config provides:

- Additional Config Sources
- Additional Converters
- Interceptors for configuration value resolution
- Relocate Configuration Properties
- Fallback Configuration Properties
- Logging
- Hide Secrets

For more information, please check the SmallRye Config documentation.