

# Quarkus - Using Kotlin

[Kotlin](#) is a very popular programming language that targets the JVM (amongst other environments). Kotlin has experienced a surge in popularity the last few years making it the most popular JVM language, except for Java of course.

Quarkus provides first class support for using Kotlin as will be explained in this guide.



This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

## Prerequisites

To complete this guide, you need:

- less than 10 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3

NB: For Gradle project setup please see below, and for further reference consult the guide in the [Gradle setup page](#).

## Creating the Maven project

First, we need a new Kotlin project. This can be done using the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=rest-kotlin-quickstart \
  -DclassName="org.acme.rest.GreetingResource" \
  -Dpath="/greeting" \
  -Dextensions="kotlin,resteasy-jsonb"
cd rest-kotlin-quickstart
```

When adding `kotlin` to the extensions list, the Maven plugin will generate a project that is properly configured to work with Kotlin. Furthermore the `org.acme.rest.GreetingResource` class is implemented as Kotlin source code (as is the case with the generated tests). The addition of

`resteasy-jsonb` in the extension list results in importing the RESTEasy/JAX-RS and JSON-B extensions.

`GreetingResource.kt` looks like this:

```
package org.acme.rest

import javax.ws.rs.GET
import javax.ws.rs.Path
import javax.ws.rs.Produces
import javax.ws.rs.core.MediaType

@Path("/greeting")
class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    fun hello() = "hello"
}
```

## Update code

In order to show a more practical example of Kotlin usage we will add a simple `data class` called `Greeting.kt` like so:

```
package org.acme.rest

data class Greeting(val message: String = "")
```

We also update the `GreetingResource.kt` like so:

```
import javax.ws.rs.GET
import javax.ws.rs.Path
import javax.ws.rs.Produces
import javax.ws.rs.core.MediaType

@Path("/greeting")
class GreetingResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    fun hello() = Greeting("hello")
}
```

With these changes in place the `/greeting` endpoint will reply with a JSON object instead of a simple String.

To make the test pass, we also need to update `GreetingResourceTest.kt` like so:

```
@QuarkusTest
class GreetingResourceTest {

    @Test
    fun testHelloEndpoint() {
        given()
            .`when`() .get("/greeting")
            .then()
                .statusCode(200)
                .body("message", equalTo("hello"))
    }
}
```

## Important Maven configuration points

The generated `pom.xml` contains the following modifications compared to its counterpart when Kotlin is not selected:

- The `quarkus-kotlin` artifact is added to the dependencies. This artifact provides support for Kotlin in the live reload mode (more about this later on)
- The `kotlin-stdlib-jdk8` is also added as a dependency.
- Maven's `sourceDirectory` and `testSourceDirectory` build properties are configured to point to Kotlin sources (`src/main/kotlin` and `src/test/kotlin` respectively)
- The `kotlin-maven-plugin` is configured as follows:

```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>
  <executions>
    <execution>
      <id>compile</id>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
    <execution>
      <id>test-compile</id>
      <goals>
        <goal>test-compile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <compilerPlugins>
      <plugin>all-open</plugin>
    </compilerPlugins>

    <pluginOptions>
      <!-- Each annotation is placed on its own line -->
      <option>all-open:annotation=javax.ws.rs.Path</option>
    </pluginOptions>
  </configuration>

  <dependencies>
    <dependency>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-maven-allopen</artifactId>
      <version>${kotlin.version}</version>
    </dependency>
  </dependencies>
</plugin>

```

The important thing to note is the use of the [all-open](#) Kotlin compiler plugin. In order to understand why this plugin is needed, first we need to note that by default all the classes generated from the Kotlin compiler are marked as **final**.

Having **final** classes however does not work well with various frameworks that need to create [Dynamic Proxies](#).

Thus, the **all-open** Kotlin compiler plugin allows us to configure the compiler to **not** mark as **final** classes that have certain annotations. In the snippet above, we have specified that classes annotated with **javax.ws.rs.Path** should not be **final**.

If your application contains classes annotated with `javax.enterprise.context.ApplicationScoped` for example, then `<option>all-open:annotation=javax.enterprise.context.ApplicationScoped</option>` needs to be added as well. Same goes for any class that needs to have a dynamic proxy created at runtime.

Future versions of Quarkus will configure the Kotlin compiler plugin in a way that will make it unnecessary to alter this configuration.

## Important Gradle configuration points

Similar to the Maven configuration, when using Gradle, the following modifications are required when Kotlin is selected:

- The `quarkus-kotlin` artifact is added to the dependencies. This artifact provides support for Kotlin in the live reload mode (more about this later on)
- The `kotlin-stdlib-jdk8` is also added as a dependency.
- The Kotlin plugin is activated, which implicitly adds `sourceDirectory` and `testSourceDirectory` build properties to point to Kotlin sources (`src/main/kotlin` and `src/test/kotlin` respectively)
- The all-open Kotlin plugin tells the compiler not to mark as final, those classes with the annotations highlighted (customize as required)
- When using native-image, the use of http (or https) protocol(s) must be declared
- An example configuration follows:

```

plugins {
    id 'java'
    id 'io.quarkus' version '1.8.0.CR1' ❶

    id "org.jetbrains.kotlin.jvm" version "{kotlin-version}" ❷
    id "org.jetbrains.kotlin.plugin.allopen" version "{kotlin-
version}" ❷
}

repositories {
    mavenCentral()
}

group = '...' // set your group
version = '1.0.0-SNAPSHOT'

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib-
jdk8:{kotlin-version}'

    ❸
    implementation enforcedPlatform('io.quarkus:quarkus-
bom:1.8.0.CR1')
    implementation 'io.quarkus:quarkus-resteasy'
    implementation 'io.quarkus:quarkus-resteasy-jsonb'
    implementation 'io.quarkus:quarkus-kotlin'

    testImplementation 'io.quarkus:quarkus-junit5'
    testImplementation 'io.rest-assured:rest-assured'
}

sourceCompatibility = '1.8'
targetCompatibility = '1.8'

test {
    useJUnitPlatform()
    exclude '**/Native*'
}

buildNative {
    enableUrlHandler = true
}

allOpen { ❹
    annotation("javax.ws.rs.Path")
    annotation("javax.enterprise.context.ApplicationScoped")
    annotation("io.quarkus.test.junit.QuarkusTest")
}

```

- ① The Quarkus plugin needs to be applied.
- ② The Kotlin plugin version needs to be specified.
- ③ We include the Quarkus BOM using Gradle's [relevant syntax](#)
- ④ The all-open configuration required, as per Maven guide above

or, if you use the Gradle Kotlin DSL:

```
import io.quarkus.gradle.tasks.QuarkusNative
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

group = "..."
version = "1.0.0-SNAPSHOT"

plugins {
    java
    id("io.quarkus") version "1.8.0.CR1" ①

    kotlin("jvm") version "{kotlin-version}" ②
    id("org.jetbrains.kotlin.plugin.allopen") version "{kotlin-version}" ②
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))

    ③
    implementation(enforcedPlatform("io.quarkus:quarkus-bom:${quarkusVersion}"))
    implementation("io.quarkus:quarkus-kotlin")
    implementation("io.quarkus:quarkus-resteasy")
    implementation("io.quarkus:quarkus-resteasy-jsonb")

    testImplementation("io.quarkus:quarkus-junit5")
    testImplementation("io.rest-assured:rest-assured")
}

tasks {
    named<QuarkusNative>("buildNative") {
        setEnableHttpRequestHandler(true)
    }
}

allOpen { ④
    annotation("javax.ws.rs.Path")
}
```

```

        annotation("javax.enterprise.context.ApplicationScoped")
        annotation("io.quarkus.test.junit.QuarkusTest")
    }

    java {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }

    val compileKotlin: KotlinCompile by tasks
    compileKotlin.kotlinOptions {
        jvmTarget = "1.8"
    }

    val compileTestKotlin: KotlinCompile by tasks
    compileTestKotlin.kotlinOptions {
        jvmTarget = "1.8"
    }

```

- ① The Quarkus plugin needs to be applied.
- ② The Kotlin plugin version needs to be specified.
- ③ We include the Quarkus BOM using Gradle's [relevant syntax](#)
- ④ The all-open configuration required, as per Maven guide above

## CDI @Inject with Kotlin

Kotlin reflection annotation processing differs from Java. You may experience an error when using CDI @Inject such as: "kotlin.UninitializedPropertyAccessException: lateinit property xxx has not been initialized"

In the example below, this can be easily solved by adapting the annotation, adding @field: Default, to handle the lack of a @Target on the Kotlin reflection annotation definition.

Alternatively, prefer the use of constructor injection which works without modification of the Java examples.



```

import javax.inject.Inject
import javax.enterprise.inject.Default
import javax.enterprise.context.ApplicationScoped

import javax.ws.rs.GET
import javax.ws.rs.Path
import javax.ws.rs.PathParam
import javax.ws.rs.Produces
import javax.ws.rs.core.MediaType

@ApplicationScoped
class GreetingService {

    fun greeting(name: String): String {
        return "hello $name"
    }
}

@Path("/")
class GreetingResource {

    @Inject
    @field: Default ❶
    lateinit var service: GreetingService

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("/greeting/{name}")
    fun greeting(@PathParam("name") name: String): String {
        return service.greeting(name)
    }
}

```

- ❶ Kotlin requires a `@field: xxx` qualifier as it has no `@Target` on the annotation definition. Add `@field: xxx` in this example. `@Default` is used as the qualifier, explicitly specifying the use of the default bean.

## Live reload

Quarkus provides support for live reloading changes made to source code. This support is also available to Kotlin, meaning that developers can update their Kotlin source code and immediately see their changes reflected.

To see this feature in action, first execute: `./mvnw compile quarkus:dev`

When executing an HTTP GET request against <http://localhost:8080/greeting>, you see a JSON message with the value `hello` as its `message` field.

Now using your favorite editor or IDE, update `GreetingResource.kt` and change the `hello` method to the following:

```
fun hello() = Greeting("hi")
```

When you now execute an HTTP GET request against <http://localhost:8080/greeting>, you should see a JSON message with the value `hi` as its `message` field.

One thing to note is that the live reload feature is not available when making changes to both Java and Kotlin source that have dependencies on each other. We hope to alleviate this limitation in the future.

## Packaging the application

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

## Kotlin and Jackson

If the `com.fasterxml.jackson.module:jackson-module-kotlin` dependency and the `quarkus-jackson` extension (or the `quarkus-resteasy-extension`) have been added to project, then Quarkus automatically registers the `KotlinModule` to the `ObjectMapper` bean (see [this](#) guide for more details).