Quarkus - Amazon Lambda

The `quarkus-amazon-lambda` extension allows you to use Quarkus to build your AWS Lambdas. Your lambdas can use injection annotations from CDI or Spring and other Quarkus facilities as you need them.

Quarkus lambdas can be deployed using the Amazon Java Runtime, or you can build a native executable and use Amazon's Custom Runtime if you want a smaller memory footprint and faster cold boot startup time.

> 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 30 minutes
* JDK 11 (AWS requires JDK 1.8 or 11)
* Apache Maven 3.6.3
* An Amazon AWS account
* AWS CLI
* AWS SAM CLI, for local testing

> For Gradle projects please see below, or for further reference consult the guide in the Gradle setup page.

Getting Started

This guide walks you through generating an example Java project via a maven archetype and deploying it to AWS.

Installing AWS bits

Installing all the AWS bits is probably the most difficult thing about this guide. Make sure that you follow all the steps for installing AWS CLI.

Creating the Maven Deployment Project

Create the Quarkus AWS Lambda maven project using our Maven Archetype.

```
mvn archetype:generate \
       -DarchetypeGroupId=io.quarkus \
       -DarchetypeArtifactId=quarkus-amazon-lambda-archetype \
       -DarchetypeVersion=1.7.0.CR1
```

If you prefer to use Gradle, you can quickly and easily generate a Gradle project via code.quarkus.io adding the `quarkus-amazon-lambda` extension as a dependency.

Copy the build.gradle, gradle.properties and settings.gradle into the above generated Maven archetype project, to follow along with this guide.

Execute: gradle wrapper to setup the gradle wrapper (recommended).

The dependency for `quarkus-test-amazon-lambda` will also need to be added to your build.gradle.

For full Gradle details see below.

Choose Your Lambda

The `quarkus-amazon-lambda` extension scans your project for a class that directly implements the Amazon `RequestHandler<?, ?>` or `RequestStreamHandler` interface. It must find a class in your project that implements this interface or it will throw a build time failure. If it finds more than one handler class, a build time exception will also be thrown.

Sometimes, though, you might have a few related lambdas that share code and creating multiple maven modules is just an overhead you don't want to do. The `quarkus-amazon-lambda` extension allows you to bundle multiple lambdas in one project and use configuration or an environment variable to pick the handler you want to deploy.

The generated project has three lambdas within it. Two that implement the `RequestHandler<?, ?>` interface, and one that implements the `RequestStreamHandler` interface. One that is used and two that are unused. If you open up `src/main/resources/application.properties` you'll see this:

```
quarkus.lambda.handler=test
```

The `quarkus.lambda.handler` property tells Quarkus which lambda handler to deploy. This can be overridden with an environment variable too.

If you look at the three generated handler classes in the project, you'll see that they are `@Named` differently.

```
@Named("test")
public class TestLambda implements RequestHandler<InputObject,
OutputObject> {
}

@Named("unused")
public class UnusedLambda implements RequestHandler<InputObject,
OutputObject> {
}

@Named("stream")
public class StreamLambda implements RequestStreamHandler {
}
```

The CDI name of the handler class must match the value specified within the
`quarkus.lambda.handler` property.

Deploy to AWS Lambda Java Runtime

There are a few steps to get your lambda running on AWS. The generated maven project contains a helpful script to create, update, delete, and invoke your lambdas for pure Java and native deployments.

Build and Deploy

Build the project using maven.

```
./mvnw clean package
```

or, if using Gradle:

```
./gradlew clean assemble
```

This will compile and package your code.

Create an Execution Role

View the Getting Started Guide for deploying a lambda with AWS CLI. Specifically, make sure you have created an `Execution Role`. You will need to define a `LAMBDA_ROLE_ARN` environment variable in your profile or console window, Alternatively, you can edit the `manage.sh` script that is generated by the build and put the role value directly there:

```
LAMBDA_ROLE_ARN="arn:aws:iam::1234567890:role/lambda-role"
```

Extra Build Generated Files

After you run the build, there are a few extra files generated by the `quarkus-amazon-lambda` extension. These files are in the the build directory: `target/` for maven, `build/` for gradle.

- `function.zip` - lambda deployment file

- `manage.sh` - wrapper around aws lamba cli calls

- `bootstrap-example.sh` - example bootstrap script for native deployments

- `sam.jvm.yaml` - (optional) for use with sam cli and local testing

- `sam.native.yaml` - (optional) for use with sam cli and native local testing

Create the function

The `target/manage.sh` script is for managing your lambda using the AWS Lambda Java runtime. This script is provided only for your convenience. Examine the output of the `manage.sh` script if you want to learn what aws commands are executed to create, delete, and update your lambdas.

`manage.sh` supports four operation: `create`, `delete`, `update`, and `invoke`.

> ℹ️ To verify your setup, that you have the AWS CLI installed, executed aws configure for the AWS access keys, and setup the `LAMBDA_ROLE_ARN` environment variable (as described above), please execute `manage.sh` without any parameters. A usage statement will be printed to guide you accordingly.

> ℹ️ If using Gradle, the path to the binaries in the `manage.sh` must be changed from `target` to `build`

To see the `usage` statement, and validate AWS configuration:

```
sh target/manage.sh
```

You can `create` your function using the following command:

```
sh target/manage.sh create
```

or if you do not have `LAMBDA_ROLE_ARN` already defined in this shell:

```
LAMBDA_ROLE_ARN="arn:aws:iam::1234567890:role/lambda-role" sh
target/manage.sh create
```

⚠️ Do not change the handler switch. This must be hardcoded to `io.quarkus.amazon.lambda.runtime.QuarkusStreamHandler::handleRequest`. This handler bootstraps Quarkus and wraps your actual handler so that injection can be performed.

If there are any problems creating the function, you must delete it with the `delete` function before re-running the `create` command.

```
sh target/manage.sh delete
```

Commands may also be stacked:

```
sh target/manage.sh delete create
```

Invoke the Lambda

Use the `invoke` command to invoke your function.

```
sh target/manage.sh invoke
```

The example lambda takes input passed in via the `--payload` switch which points to a json file in the root directory of the project.

The lambda can also be invoked locally with the SAM CLI like this:

```
sam local invoke --template target/sam.jvm.yaml --event
payload.json
```

If you are working with your native image build, simply replace the template name with the native version:

```
sam local invoke --template target/sam.native.yaml --event
payload.json
```

Update the Lambda

You can update the Java code as you see fit. Once you've rebuilt, you can redeploy your lambda by

executing the `update` command.

```
sh target/manage.sh update
```

Deploy to AWS Lambda Custom (native) Runtime

If you want a lower memory footprint and faster initialization times for your lambda, you can compile your Java code to a native executable. Just make sure to rebuild your project with the `-Pnative` switch.

For Linux hosts execute:

```
mvn package -Pnative
```

or, if using Gradle:

```
./gradlew build -Dquarkus.package.type=native
```

> ℹ️ If you are building on a non-Linux system, you will need to also pass in a property instructing quarkus to use a docker build as Amazon Lambda requires linux binaries. You can do this by passing this property to your Maven build: `-Dnative-image.docker-build=true`, or for Gradle: `--docker-build=true`. This requires you to have docker installed locally, however.

```
./mvnw clean install -Pnative -Dnative-image.docker-build=true
```

or, if using Gradle:

```
./gradlew build -Dquarkus.package.type=native
-Dquarkus.native.container-build=true
```

Either of these commands will compile and create a native executable image. It also generates a zip file `target/function.zip`. This zip file contains your native executable image renamed to `bootstrap`. This is a requirement of the AWS Lambda Custom (Provided) Runtime.

The instructions here are exactly as above with one change: you'll need to add `native` as the first parameter to the `manage.sh` script:

```
sh target/manage.sh native create
```

As above, commands can be stacked. The only requirement is that `native` be the first parameter should you wish to work with native image builds. The script will take care of the rest of the details necessary to manage your native image function deployments.

Examine the output of the `manage.sh` script if you want to learn what aws commands are executed to create, delete, and update your lambdas.

One thing to note about the create command for native is that the `aws lambda create-function` call must set a specific environment variable:

```
--environment 'Variables={DISABLE_SIGNAL_HANDLERS=true}'
```

Examine the POM and Gradle build

There is nothing special about the POM other than the inclusion of the `quarkus-amazon-lambda` and `quarkus-test-amazon-lambda` extensions as a dependencies. The extension automatically generates everything you might need for your lambda deployment.

> ℹ In previous versions of this extension you had to set up your pom or gradle to zip up your executable for native deployments, but this is not the case anymore.

Gradle build

Similarly for Gradle projects, you also just have to add the `quarkus-amazon-lambda` and `quarkus-test-amazon-lambda` dependencies. The extension automatically generates everything you might need for your lambda deployment.

Example Gradle dependencies:

```
dependencies {
    implementation
enforcedPlatform("${quarkusPlatformGroupId}:${quarkusPlatformArtifa
ctId}:${quarkusPlatformVersion}")
    implementation 'io.quarkus:quarkus-resteasy'
    implementation 'io.quarkus:quarkus-amazon-lambda'

    testimplementation  "io.quarkus:quarkus-test-amazon-lambda"

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

Integration Testing

The Quarkus Amazon Lambda extension has a matching test framework that provides functionality to execute standard JUnit tests on your AWS Lambda function, via the integration layer that Quarkus provides. This is true for both JVM and native modes. It provides similar functionality to the SAM CLI, without the overhead of Docker.

To illustrate, the project generated by the Maven archetype, generates a JUnit test for the `RequestHandler<?, ?>` implementation, which is shown below. The test replicates the execution environment, for the function that is selected for invocation, as described above.

To use the integration tests in your project there is a required property, in `src/test/resources/application.properties`. If not included, the integration tests will be in a constant loop.

```
quarkus.lambda.enable-polling-jvm-mode=true
```

> ℹ️ If you are following along with the example Maven archetype project for AWS Lambda in this guide, it includes the required property `quarkus.lambda.enable-polling-jvm-mode=true` in the test `application.properties`.

```java
@QuarkusTest
public class LambdaHandlerTest {

    @Test
    public void testSimpleLambdaSuccess() throws Exception {
        InputObject in = new InputObject();
        in.setGreeting("Hello");
        in.setName("Stu");

        OutputObject out = LambdaClient.invoke(OutputObject.class, in);

        Assertions.assertEquals("Hello Stu", out.getResult());
        Assertions.assertTrue(out.getRequestId().matches("aws-request-\\d"), "Expected requestId as 'aws-request-<number>'");
    }
}
```

Similarly, if you are using a `RequestStreamHandler` implementation, you can add a matching JUnit test, like below, which aligns to the generated `StreamLambda` class in the generated project.

Obviously, these two types of tests are mutually exclusive. You must have a test that corresponds to the implemented AWS Lambda interfaces, whether `RequestHandler<?, ?>` or `RequestStreamHandler`.

Two versions of the Test for `RequestStreamHandler` are presented below. You can use either, depending on the needs of your Unit test. The first is obviously simpler and quicker. Using Java streams can require more coding.

```java
@QuarkusTest
public class LambdaStreamHandlerTest {

    private static Logger LOG =
Logger.getLogger(LambdaStreamHandlerTest.class);

    @Test
    public void testSimpleLambdaSuccess() throws Exception {
        String out = LambdaClient.invoke(String.class,
"lowercase");
        Assertions.assertEquals("LOWERCASE", out);
    }

    @Test
    public void testInputStreamSuccess() {
        try {
            String input = "{ \"name\": \"Bill\", \"greeting\":
\"hello\"}";
            InputStream inputStream = new
ByteArrayInputStream(input.getBytes());
            ByteArrayOutputStream outputStream = new
ByteArrayOutputStream();

            LambdaClient.invoke(inputStream, outputStream);

            ByteArrayInputStream out = new
ByteArrayInputStream(outputStream.toByteArray());
            StringBuilder response = new StringBuilder();
            int i = 0;
            while ((i = out.read()) != -1) {
                response.append((char)i);
            }


Assertions.assertTrue(response.toString().contains("BILL"));
        } catch (Exception e) {
            Assertions.fail(e.getMessage());
        }
    }

}
```

If your code uses CDI injection, this too will be executed, along with mocking functionality, see the Test Guide for more details.

To add JUnit functionality for native tests, add the `@NativeImageTest` annotation to a subclass of your test class, which will execute against your native image, and can be leveraged in an IDE.

Testing with the SAM CLI

The AWS SAM CLI allows you to run your lambdas locally on your laptop in a simulated Lambda environment. This requires docker to be installed. This is an optional approach should you choose to take advantage of it. Otherwise, the Quarkus JUnit integration should be sufficient for most of your needs.

A starter template has been generated for both JVM and native execution modes.

Run the following SAM CLI command to locally test your lambda function, passing the appropriate SAM `template`. The `event` parameter takes any JSON file, in this case the sample `payload.json`.

> **ℹ** If using Gradle, the path to the binaries in the YAML templates must be changed from `target` to `build`

```
sam local invoke --template target/sam.jvm.yaml --event
payload.json
```

The native image can also be locally tested using the `sam.native.yaml` template:

```
sam local invoke --template target/sam.native.yaml --event
payload.json
```

Modifying `function.zip`

The are times where you may have to add some additions to the `function.zip` lambda deployment that is generated by the build. To do this create a `zip.jvm` or `zip.native` directory within `src/main`. Create `zip.jvm/` if you are doing a pure Java lambda. `zip.native/` if you are doing a native deployment.

Any you files and directories you create under your zip directory will be included within `function.zip`

Custom `bootstrap` script

There are times you may want to set a specific system properties or other arguments when lambda invokes your native quarkus lambda deployment. If you include a `bootstrap` script file within `zip.native`, the quarkus extension will automatically rename the executable to `runner` within `function.zip` and set the unix mode of the `bootstrap` script to executable.

> ℹ️ The native executable must be referenced as `runner` if you include a custom `bootstrap` script.

The extension generates an example script within `target/bootstrap-example.sh`.

Tracing with AWS XRay and GraalVM

If you are building native images, and want to use AWS X-Ray Tracing with your lambda you will need to include `quarkus-amazon-lambda-xray` as a dependency in your pom. The AWS X-Ray library is not fully compatible with GraalVM so we had to do some integration work to make this work.

In addition, remember to enable the AWS X-Ray tracing parameter in `manage.sh`, in the `cmd_create()` function. This can also be set in the AWS Management Console.

```
--tracing-config Mode=Active
```

For the sam template files, add the following to the YAML function Properties.

```
Tracing: Active
```

AWS X-Ray does add many classes to your distribution, do ensure you are using at least the 256MB AWS Lambda memory size. This is explicitly set in `manage.sh cmd_create()`. Whilst the native image potentially can always use a lower memory setting, it would be recommended to keep the setting the same, especially to help compare performance.

Using HTTPS or SSL/TLS

If your code makes HTTPS calls, such as to a micro-service (or AWS service), you will need to add configuration to the native image, as GraalVM will only include the dependencies when explicitly declared. Quarkus, by default enables this functionality on extensions that implicitly require it. For further information, please consult the Quarkus SSL guide

Open src/main/resources/application.properties and add the following line to enable SSL in your native image.

```
quarkus.ssl.native=true
```

Using the AWS Java SDK v2

> ℹ️ Quarkus now has extensions for DynamoDB, S3, SNS and SQS (more coming). Please check those guides on how to use the various AWS Services with Quarkus, as opposed to wiring manually like below.

With minimal integration, it is possible to leverage the AWS Java SDK v2, which can be used to invoke services such as SQS, SNS, S3 and DynamoDB.

For native image, however the URL Connection client must be preferred over the Apache HTTP Client when using synchronous mode, due to issues in the GraalVM compilation (at present).

Add `quarkus-jaxb` as a dependency in your Maven `pom.xml`, or Gradle `build.gradle` file.

You must also force your AWS service client for SQS, SNS, S3 et al, to use the URL Connection client, which connects to AWS services over HTTPS, hence the inclusion of the SSL enabled property, as described in the Using HTTPS or SSL/TLS section above.

```
// select the appropriate client, in this case SQS, and
// insert your region, instead of XXXX, which also improves startup
time over the default client
  client =
SqsClient.builder().region(Region.XXXX).httpClient(software.amazon.
awssdk.http.urlconnection.UrlConnectionHttpClient.builder().build()
).build();
```

For Maven, add the following to your `pom.xml`.

```
    <properties>
        <aws.sdk2.version>2.10.69</aws.sdk2.version>
    </properties>

    <dependencyManagement>
        <dependencies>

            <dependency>
                <groupId>software.amazon.awssdk</groupId>
                <artifactId>bom</artifactId>
                <version>${aws.sdk2.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>

        </dependencies>
    </dependencyManagement>
    <dependencies>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>url-connection-client</artifactId>
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
```

```xml
            <artifactId>apache-client</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>commons-logging</groupId>
                    <artifactId>commons-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <!-- sqs/sns/s3 etc -->
            <artifactId>sqs</artifactId>
            <exclusions>
                <!-- exclude the apache-client and netty client -->
                <exclusion>
                    <groupId>software.amazon.awssdk</groupId>
                    <artifactId>apache-client</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>software.amazon.awssdk</groupId>
                    <artifactId>netty-nio-client</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>commons-logging</groupId>
                    <artifactId>commons-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.jboss.logging</groupId>
            <artifactId>commons-logging-jboss-logging</artifactId>
            <version>1.0.0.Final</version>
        </dependency>
    </dependencies>
```

> 🛈 if you see `java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty` or similar SSL error, due to the current status of GraalVM, there is some additional work to bundle the `function.zip`, as below. For more information, please see the Quarkus Native SSL Guide.

Additional requirements for client SSL

The native executable requires some additional steps to enable client ssl that S3 and other aws libraries need.

1. A custom `bootstrap` script

2. `libsunec.so` must be added to `function.zip`

3. `cacerts` must be added to `function.zip`

To do this, first create a directory `src/main/zip.native/` with your build. Next create a shell script file called `bootstrap` within `src/main/zip.native/`, like below. An example is create automatically in your build folder (target or build), called `bootstrap-example.sh`

```bash
#!/usr/bin/env bash

./runner -Djava.library.path=./
-Djavax.net.ssl.trustStore=./cacerts
```

Additional set `-Djavax.net.ssl.trustStorePassword=changeit` if your `cacerts` file is password protected.

Next you must copy some files from your GraalVM distribution into `src/main/zip.native/`.

> ℹ️ GraalVM versions can have different paths for these files, and whether you using the Java 8 or 11 version. Adjust accordingly.

```
cp $GRAALVM_HOME/lib/libsunec.so $PROJECT_DIR/src/main/zip.native/
cp $GRAALVM_HOME/lib/security/cacerts $PROJECT_DIR/src/main/zip.native/
```

Now when you run the native build all these files will be included within `function.zip`

> ℹ️ If you are using a Docker image to build, then you must extract these files from this image.

To extract the required ssl, you must start up a Docker container in the background, and attach to that container to copy the artifacts.

First, let's start the GraalVM container, noting the container id output.

```
docker run -it -d --entrypoint bash quay.io/quarkus/ubi-quarkus-native-image:20.1.0-java11

# This will output a container id, like
6304eea6179522aff69acb38eca90bedfd4b970a5475aa37ccda3585bc2abdde
# Note this value as we will need it for the commands below
```

First, libsunec.so, the C library used for the SSL implementation:

```
docker cp {container-id-from-
above}:/opt/graalvm/jre/lib/amd64/libsunec.so src/main/zip.native/
```

Second, cacerts, the certificate store. You may need to periodically obtain an updated copy, also.

```
docker cp {container-id-from-
above}:/opt/graalvm/jre/lib/security/cacerts src/main/zip.native/
```

Your final archive will look like this:

```
jar tvf target/function.zip

    bootstrap
    runner
    cacerts
    libsunec.so
```

Amazon Alexa Integration

To use Alexa with Quarkus native, please add the following extension.

```
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-amazon-alexa</artifactId>
        </dependency>
```

Create your Alexa handler, as normal, by sub-classing the abstract `com.amazon.ask.SkillStreamHandler`, and add your request handler implementation.

That's all there is to it!