Quarkus - Amazon Lambda

The `quarkus-amazon-lambda` extension allows you to use Quarkus to build your Amazon 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 1.8 (AWS requires JDK 1.8)
- Apache Maven 3.5.3+
- An Amazon AWS account
- AWS CLI
- AWS SAM CLI, for local testing

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=1.3.0.Alpha1
```

Choose Your Lambda

The `quarkus-amazon-lambda` extension scans your project for a class that implements the Amazon `RequestHandler` 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 two lambdas within it. One that is used and one that is 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 two 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> {
}
```

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.

```
mvn clean install
```

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 copy and paste the `Role Arn` into the scripts generated by the maven archetype. To use the provided scripts, you will either need to define a `LAMBDA_ROLE_ARN` environment variable in your profile or define it prior to executing the scripts like this:

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

Create the function

The `manage.sh` script is for managing your lambda using the AWS Lambda Java runtime. This script is provided only for your convenience. Should you choose to use Amazon provided tools, you can freely skip these sections and consult the appropriate Amazon documentation.

`manage.sh` supports four operation: `create`, `delete`, `update`, and `invoke`. You can create your function using the following command:

```
sh 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
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 manage.sh delete
```

Commands may also be stacked:

```
sh manage.sh delete create
```

Invoke the Lambda

Use the `invoke` command to invoke your function.

```
sh 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. If you want to see the output of the lambda, open the `response.txt` file. The lambda can also be invoked locally like this:

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

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

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 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. 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`. This requires you to have docker installed locally, however.

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

This 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 Amazon Lambda Custom Runtime. If you are building on Linux, the `native-image.docker-build` property is mildly redundant but its primary cost is a slightly longer build cycle.

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 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 POM

If you want to adapt an existing project to use Quarkus's Amazon Lambda extension, there are a couple of things you need to do. Take a look at the generated example project to get an example of what you need to adapt.

1. Include the `quarkus-amazon-lambda` extension as a pom dependency

2. Configure Quarkus to build an `uber-jar` (via quarkus.package.uber-jar=true in the application.properties)

3. If you are doing a native image build, Amazon requires you to rename your executable to `bootstrap` and zip it up. Notice that the `pom.xml` uses the `maven-assembly-plugin` to perform this requirement.

NB: With gradle, to build the uber-jar execute: ./gradlew quarkusBuild --uber-jar

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, dev mode 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:

```
sam local invoke --template sam.jvm.yaml --event {json test file}
```

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

```
sam local invoke --template sam.native.yaml --event {json test
file}
```