

Quarkus - Amazon Lambda with RESTEasy, Undertow, or Vert.x Web

The `quarkus-amazon-lambda-http` extension allows you to write microservices with RESTEasy (JAX-RS), Undertow (servlet), Vert.x Web, or [Fungy HTTP](#) and make these microservices deployable as an Amazon Lambda using [Amazon's API Gateway](#) and [Amazon's SAM framework](#).

You can deploy your Lambda as a pure Java jar, or you can compile your project to a native image and deploy that for a smaller memory footprint and 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 SAM CLI](#)

Getting Started

This guide walks you through generating an example Java project via a maven archetype. Later on it walks through the structure of the project so you can adapt any existing projects you have to use Amazon Lambda.

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 SAM 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-http-archetype \  
    -DarchetypeVersion=1.7.4.Final
```

Build and Deploy

Build the project using maven.

```
./mvnw clean install
```

This will compile the code and run the unit tests included within the generated project. Unit testing is the same as any other Java project and does not require running on Amazon. Quarkus dev-mode is also available with this extension.

If you want to build for native too, make sure you have GraalVM installed correctly and just add a **native** property to the build

```
./mvnw clean install -Dnative
```



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 -Dnative -Dnative-image.docker-build=true
```

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
- **sam.jvm.yaml** - sam cli deployment script
- **sam.native.yaml** - sam cli deployment script for native

Simulate Amazon Lambda Deployment

The AWS SAM CLI allows you to run your lambda's locally on your laptop in a simulated Lambda environment. This requires docker to be installed (see their install docs). After you have built your maven project, execute this command

```
sam local start-api --template target/sam.jvm.yaml
```

This will start a docker container that mimics Amazon's Lambda's deployment environment. Once the environment is started you can invoke the example lambda in your browser by going to

<http://127.0.0.1:3000/hello>

In the console you'll see startup messages from the lambda. This particular deployment starts a JVM and loads your lambda as pure Java.

If you want to deploy a native executable of your lambda, use a different yaml template that is provided in your generated project:

```
sam local start-api --template target/sam.native.yaml
```

Deploy to AWS

There are a few steps to get your lambda running on AWS.

Package your deployment.

```
sam package --template-file target/sam.jvm.yaml --output-template  
-file packaged.yaml --s3-bucket <YOUR_S3_BUCKET>
```

Type the simple name of your S3 bucket you created during. If you've built a native executable, replace `sam.jvm.yaml` with `sam.native.yaml`.

Deploy your package

```
sam deploy --template-file packaged.yaml --capabilities  
CAPABILITY_IAM --stack-name <YOUR_STACK_NAME>
```

The stack name can be anything you want.

Debugging AWS Deployment Problems

If `sam deploy`, run the `describe-stack-events` command to get information about your deployment and what happened.

```
aws cloudformation describe-stack-events --stack-name  
<YOUR_STACK_NAME>
```

One common issue that you may run across is that your S3 bucket has to be in the same region as Amazon Lambda. Look for this error from `describe-stack-events` output:

```
Error occurred while GetObject. S3 Error Code:  
AuthorizationHeaderMalformed. S3 Error Message:  
The authorization header is malformed; the region 'us-east-1' is  
wrong; expecting 'us-east-2'  
(Service: AWSLambdaInternal; Status Code: 400; Error Code:  
InvalidParameterValueException;  
Request ID: aeefcf978-ad2a-4b53-9ffe-cea3fcd0f868)
```

The above error is stating that my S3 bucket should be in `us-east-2`, not `us-east-1`. To fix this error you'll need to create an S3 bucket in that region and redo steps 1 and 2 from above.

Another annoying this is that if there is an error in deployment, you also have to completely delete it before trying to deploy again:

```
aws cloudformation delete-stack --stack-name <YOUR_STACK_NAME>
```

Execute your REST Lambda on AWS

To get the root URL for your service, type the following command and see the following output:

```
aws cloudformation describe-stacks --stack-name <YOUR_STACK_NAME>
```

It should give you something like the following output:

```
{
  "Stacks": [
    {
      "StackId": "arn:aws:cloudformation:us-east-1:502833056128:stack/QuarkusNativeRestExample2/b35b0200-f685-11e9-aaa0-0e8cd4caae34",
      "DriftInformation": {
        "StackDriftStatus": "NOT_CHECKED"
      },
      "Description": "AWS Serverless Quarkus HTTP - io.demo::rest-example",
      "Tags": [],
      "Outputs": [
        {
          "Description": "URL for application",
          "ExportName": "RestExampleNativeApi",
          "OutputKey": "RestExampleNativeApi",
          "OutputValue": "https://234234234.execute-api.us-east-1.amazonaws.com/Prod/"
        }
      ]
    }
  ]
}
```

The **OutputValue** attribute is the root URL for your lambda. Copy it to your browser and add **hello** at the end.



Responses for binary types will be automatically encoded with base64. This is different than the behavior using **quarkus:dev** which will return the raw bytes. Amazon's API has additional restrictions requiring the base64 encoding. In general, client code will automatically handle this encoding but in certain custom situations, you should be aware you may need to manually manage that encoding.

Examine the POM

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



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.

Also, at least in the generated maven archetype **pom.xml**, the **quarkus-resteasy**, **quarkus-vertx-web**, and **quarkus-underdow** dependencies are all optional. Pick which http framework(s) you want to use (JAX-RS, Vertx Web, and/or Servlet) and remove the other dependencies to shrink your deployment.

Examine sam.yaml

The `sam.yaml` syntax is beyond the scope of this document. There's a couple of things to note though that are particular to the `quarkus-amazon-lambda-http` extension.

Amazon's API Gateway assumes that HTTP response bodies are text unless you explicitly tell it which media types are binary through configuration. To make things easier, the Quarkus extension forces a binary (base 64) encoding of all HTTP response messages and the `sam.yaml` file must configure the API Gateway to assume all media types are binary:

```
Globals:
  Api:
    EndpointConfiguration: REGIONAL
    BinaryMediaTypes:
      - "*/*"
```

Another thing to note is that for pure Java lambda deployments, do not change the Lambda handler name.

```
Properties:
  Handler:
io.quarkus.amazon.lambda.runtime.QuarkusStreamHandler::handleRequest
  Runtime: java8
```

This particular handler handles all the intricacies of integrating with the Quarkus runtime. So you must use that handler.

Finally, there's an environment variable that must be set for native GraalVM deployments. If you look at `sam.native.yaml` you'll see this:

```
Environment:
  Variables:
    DISABLE_SIGNAL_HANDLERS: true
```

This environment variable resolves some incompatibilities between Quarkus and the Amazon Lambda Custom Runtime environment.

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.