Getting Started With Quarkus Serverless Functions

CONTENTS

DANIEL OH
SR. PRINCIPAL TECHNICAL MARKETING MANAGER, RED HAT

Since the inception of Java in 1995, Java frameworks have evolved for higher performance, new features, and increased support of business applications. Over the years, enterprises have also adopted several infrastructures — from physical servers to virtual machines and cloud environments — to run Java applications. Continuous availability is the top priority of these applications; they need to be stable, reliable, and performant regardless of the workload. Therefore, most enterprises have a huge burden with the high costs of maintaining infrastructure resources (e.g., CPU, memory, disk, networking), even if the resource utilization is less than 50%.

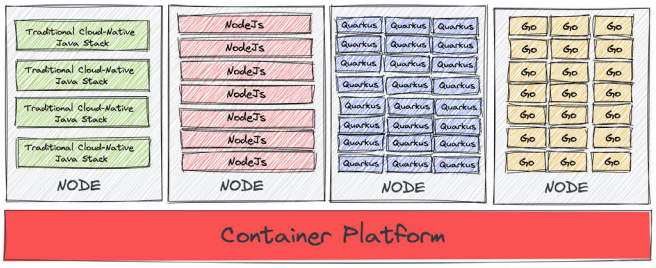
Serverless architectures were designed to provide an efficient solution to align overprovisioning and underprovisioning resources with actual workloads regardless of physical servers, virtual machines, and cloud deployments. However, these traits and benefits are challenging to realize with traditional Java microservices like Spring Boot. In many cases, traditional Java frameworks are too heavyweight and slow for serverless deployment to the cloud, especially deployment on Kubernetes.

What if Java developers could continue using the same Java frameworks to build traditional cloud-native microservices as well as new serverless functions? This approach reduces the learning curve for new serverless application frameworks.

WHAT IS QUARKUS?

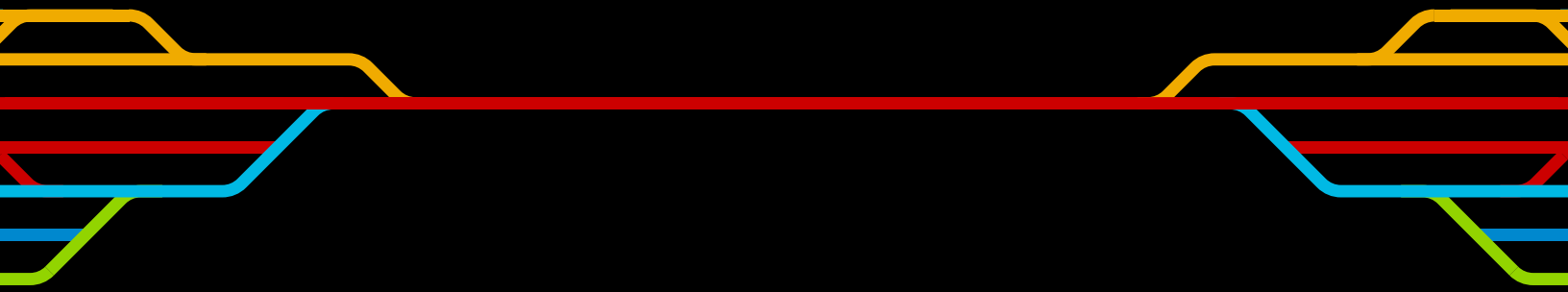
Quarkus rethinks Java, using a closed-world approach to building and running it. It has turned Java into a runtime that's comparable to Go and Node.js. Quarkus provides almost 450 extensions that integrate enterprise capabilities such as data management, serverless functions, continuous testing, web rendering, messaging, security, cloud deployment, and business process automation. Quarkus brings Java developers onto the serverless function development journey with immutable infrastructure based on containers and Kubernetes. As shown in Figure 1, developers can deploy 10 times more native Quarkus applications on Kubernetes through near-instant scale-up and high-density utilization similar to Go.

Figure 1: Quarkus Java serverless and containers



Red Hat Developer

Build here.
Go anywhere.



Join now

GET STARTED WITH QUARKUS SERVERLESS FUNCTIONS

This Refcard covers a quick tutorial for how to scaffold a new Java serverless function project with Quarkus. Then you will learn how to optimize the serverless function, make it portable across multiple serverless platforms, and bind CloudEvents on Kubernetes.

CREATE A QUARKUS SERVERLESS PROJECT

If you haven't created your Kubernetes cluster already, you can use open-source Minikube, OpenShift Kubernetes Distribution (OKD), or Red Hat OpenShift, which is built on Kubernetes with full-stack automated operations and self-service provisioning for developers. Red Hat OpenShift also enables developers to create serverless applications based on the open-source Knative project. This Refcard uses a Red Hat OpenShift cluster for the deployment environment. See the installation guides for OpenShift clusters and Serverless Operator for more information.

The following Maven command scaffolds a new Quarkus project (`quarkus-serverless-examples`) that includes a simple RESTful API. This project also installs a `quarkus-openshift` extension that you'll use to deploy the Quarkus application to the OpenShift cluster:

```
$ mvn io.quarkus:quarkus-maven-
plugin:2.2.3.Final:create \-DprojectGroupId=org.
acme \-DprojectArtifactId=quarkus-
serverless-examples \-Dextensions="openshift"
\-DclassName="org.acme.getting.started.
GreetingResource"
```

RUN SERVERLESS FUNCTIONS LOCALLY

As always, the first step to develop an application on Quarkus is to run Quarkus development mode. Run the following Maven command:

```
$ ./mvnw quarkus:dev
```

Note: Be sure to continue using Quarkus dev mode for live coding. This enables you to avoid recompiling, redeploying the application, and restarting the Quarkus runtime while you change the code.

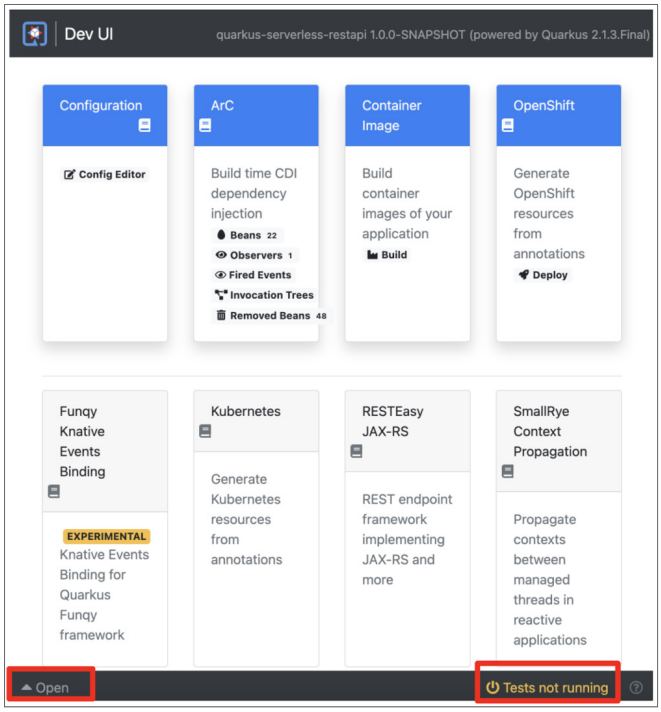
TEST SERVERLESS FUNCTIONS CONTINUOUSLY

When developing a serverless function, you have to test your code before it's ready for production. For years, a dedicated quality assurance (QA) team has been engaged in this phase to ensure business requirements using external continuous integration (CI) tools, and it is still typically done today.

However, Quarkus enables developers to run unit tests automatically while their code is both running and as it changes. Quarkus provides this continuous testing feature through the command-line interface (CLI) and the DEV UI.

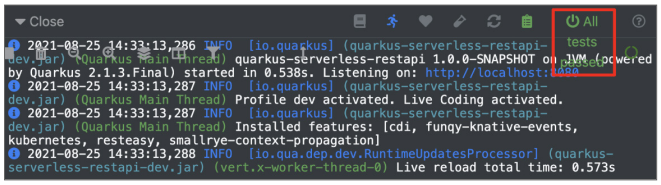
Open a new web browser to access the DEV UI (http://localhost:8080/q/dev/), as shown in Figure 2.

Figure 2: Quarkus DEV UI



Continuous testing is not running by default when a Quarkus application begins. To start it, click `Tests not running` on the bottom-right of the DEV UI. You can also open a web terminal by clicking `Open` on the left-hand side of the DEV UI. An example test result is shown in Figure 3.

Figure 3: Quarkus DEV UI — test passed



Let's invoke the REST API using the `curl` command. You can also access the endpoint via a web browser. The output should be `Hello RESTEasy`:

```
$ curl localhost:8080/hello
Hello RESTEasy
```

Update the `hello` method in the `src/main/java/GreetingResource.java` file to modify the return text:

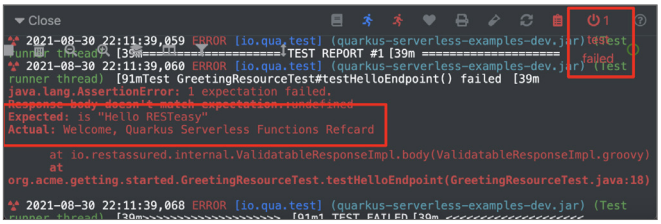
```
public String hello() {
  return "Welcome, Quarkus Serverless Functions
Refcard";
}
```

You will see the new output when you invoke the REST API again:

```
$ curl localhost:8080/hello
Quarkus Serverless Functions Refcard
```

The continuous test will fail, as shown in Figure 4. To fix it, update the return text in the test code to match the logic code (`Welcome, Quarkus Serverless Functions Refcard`).

Figure 4: Quarkus DEV UI — test failed



You can rerun all test cases implemented in the `src/test/java/` directory. This feature ensures business requirements are correctly developed in the development phase without integrating an external CI tool in the test phase.

There's not a big difference between normal microservices and serverless functions. A benefit of Quarkus is that it enables developers to use any standard microservices for deploying to Kubernetes/OpenShift as a serverless function.

DEPLOY FUNCTIONS TO A KNATIVE SERVICE

If you haven't already, create a project in OpenShift named `quarkus-serverless-examples` to deploy the Quarkus serverless function. To generate Knative and Kubernetes resources, you need to add the following Quarkus variables in `src/main/resources/application.properties`:

```
quarkus.container-image.group=quarkus-serverless-
examples <1>
quarkus.container-image.registry=image-registry.
openshift-image-registry.svc:5000 <2>
quarkus.kubernetes-client.trust-certs=true <3>
quarkus.kubernetes.deployment-target=knative <4>
quarkus.kubernetes.deploy=true <5>
quarkus.openshift.build-strategy=docker <6>
```

Note: Make sure to log in to the OpenShift cluster using the `oc` login command to access this project.

The following are descriptions explaining the six variables above:

<1> Define a project (image group) name where you deploy a serverless application image.

<2> Define a container registry where a serverless application container image pushes.

<3> Use only if you are using self-signed certs (in this example, we are using them).

<4> Generate a Knative resource file (e.g., `knative.json` and `knative.yaml`).

<5> Enable a Kubernetes/OpenShift deployment when the container image build is finished.

<6> Use the Docker build strategy.

Run the following command to build the serverless application, then deploy it directly to the OpenShift cluster:

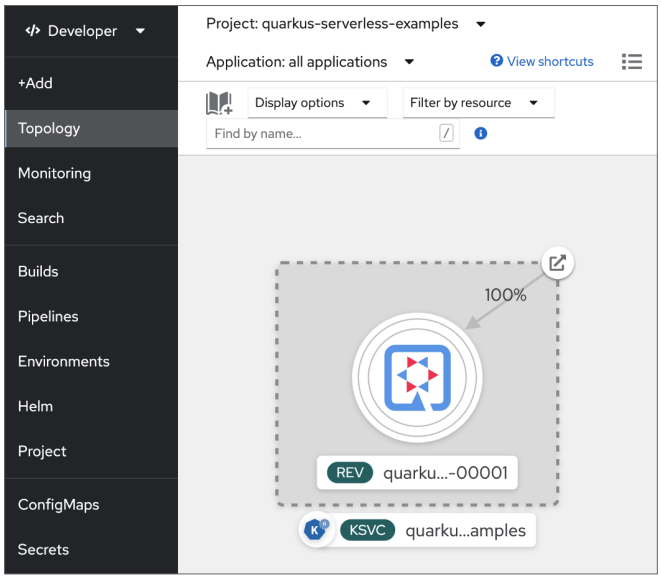
```
$ ./mvnw clean package -DskipTests
```

The output should end with `BUILD SUCCESS`. Add a Quarkus label to the Knative service that shows the Quarkus logo inside a running pod using the following `oc` command:

```
$ oc label rev/quarkus-serverless-examples-00001
app.openshift.io/runtime=quarkus --overwrite
```

Access the *Topology* view in the Developer perspective menu of the OpenShift web console to confirm if your application deployed. The serverless function pod might be scaled down to zero (white-line circle), as shown in Figure 5.

Figure 5: Quarkus serverless function in OpenShift



Retrieve a route URL of the serverless function using the following `oc` command:

```
$ oc get rt/quarkus-serverless-examples
NAME    URL    READY    REASON
quarkus-serverless[...]  http://
quarkus[...].SUBDOMAIN  True
```

Access the route URL with a `curl` command:

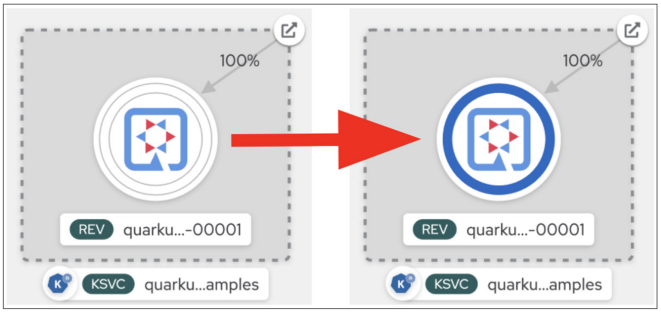
```
$ curl http://quarkus-serverless-[...].SUBDOMAIN/
hello
```

You will see the same result as you did locally. It will take a few seconds to scale up the pod:

```
Welcome, Quarkus Serverless Functions Refcard
```

Going back to the *Topology* view in the OpenShift cluster, the serverless function (Knative service) scaled up automatically, as shown in Figure 6.

Figure 6: Scale up Quarkus serverless function in OpenShift



Note: The Knative service pod will go down to zero again in 30 seconds (default setting).

MAKE SERVERLESS FUNCTIONS RUN FASTER WITH GRAALVM

Quarkus enables developers to build a native executable file with performance advantages, including fast boot time and small resident set size (RSS) memory, for near-instant scale-up and high-density memory utilization, as compared to traditional cloud-native Java frameworks.

Before we build a native executable file, let's find out how long the existing serverless functions took to start up. When you access the running Quarkus pod's logs in OpenShift, you will see the JVM (HotSpot) serverless function running as the Knative service:

```
...
[io.quarkus] (main) quarkus-serverless-examples
1.0.0-SNAPSHOT on JVM (powered by Quarkus
2.1.4.Final) started in 5.984s.
...
```

Quarkus uses GraalVM to build a native executable. You can choose any GraalVM distribution, such as Oracle GraalVM Community Edition (CE) and Mandrel (downstream distribution of Oracle GraalVM CE). Mandrel is designed to support building Quarkus-native executables on OpenJDK 11.

Open `pom.xml` to see the native profile in the Maven project. You'll use it to build a native executable file:

```
<profiles>
   <profile>
      <id>native</id>
      <properties>
         <quarkus.package.type>native</quarkus.
package.type>
      </properties>
   </profile>
</profiles>
```

Note: If you haven't already installed a GraalVM or Mandrel distribution locally, pull the Mandrel container image using your local container engine (e.g., Docker). This makes a native executable image on any Linux-supported platform.

Execute one of the following Maven commands to build a native executable image.

Using Docker:

```
$ ./mvnw package -Pnative \ -Dquarkus.native.
container-build=true
```

Using Podman:

```
$ ./mvnw package -Pnative \ -Dquarkus.native.
container-build=true \ -Dquarkus.native.container-
runtime=podman
```

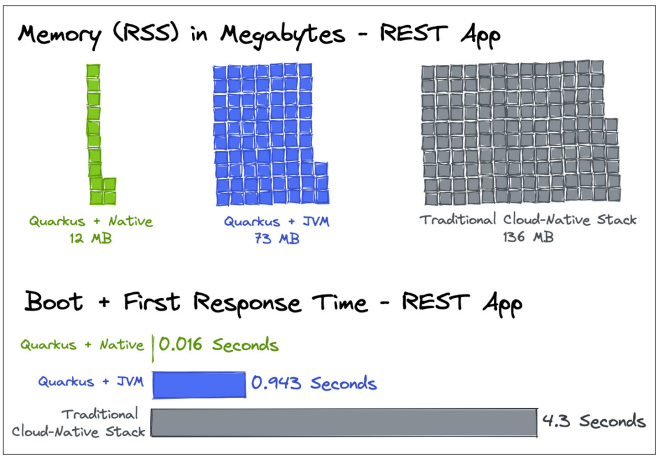
The output should end with `BUILD SUCCESS`. Access the running Quarkus pod's logs once again in OpenShift — you will see that the `native` serverless function is running as the Knative service:

```
...
[io.quarkus] (main) quarkus-serverless-examples
1.0.0-SNAPSHOT native (powered by Quarkus
2.1.4.Final) started in 0.013s.
...
```

That's **13** milliseconds to start up 48 times faster. Note that the start-up time might be different in your environment.

Figure 7 (on the following page) compares the scale-up speed and memory utilization between Quarkus and traditional cloud-native Java stacks.

Figure 7: Quarkus performance report



MAKE PORTABLE FUNCTIONS ACROSS SERVERLESS PLATFORMS

Multi- and hybrid-cloud strategies have significantly influenced the portability of application development and deployment on heterogeneous platforms — from public managed services to open-source projects. This tendency is also reflected in the serverless technology stack, which can present new challenges to developers in choosing the right frameworks and tools.

View the CNCF's Serverless Landscape here: https://landscape.cncf.io/serverless

These challenges may continue even after selecting a new serverless development framework because developers will likely still need to learn additional technologies (e.g., APIs, CLI tools, SDKs, RBAC policies) that depend on the serverless platforms where they deploy applications.

Quarkus solved this problem with Funqy extensions that enable developers to implement a serverless function. Then the function can be deployed without code changes to multiple serverless runtime environments such as AWS Lambda, Azure Functions, Google Cloud Platform, and Knative Events. Developers can spend less time and effort on learning new serverless technologies.

ADD A QUARKUS FUNQY EXTENSION TO YOUR PROJECT

Run the following Maven command to add a `quarkus-funqy-http` extension for using Funqy HTTP binding:

```
$ ./mvnw quarkus:add-extension -Dextensions="io.
quarkus:quarkus-funqy-http"
```

Update the `src/main/java/GreetingResource.java` file to make a simpler, but similar, portable serverless function. As you can see, the `@Funq` annotation exposes the `hello` method as a serverless function based on the Funqy API. The function name is equivalent to the method name (`hello`) by default. The lines of code (LOC) are also reduced by half:

```
package org.acme.getting.started;

import io.quarkus.funqy.Funq;

public class GreetingResource {
   @Funq
   public String hello() {
      return "Welcome, Quarkus Serverless
Functions Refcard";
   }
}
```

Now you can run this serverless function locally in dev mode using `./mvnw quarkus:dev`, which also enables continuous testing.

Then the application can be deployed to OpenShift via `./mvnw package -DskipTests` as you did previously. You will have the new output, `Welcome, Quarkus Serverless Functions Refcard`, when you access the endpoint (`/hello`) after you deploy it once again.

DEPLOY A QUARKUS FUNQY APPLICATION TO AWS LAMBDA

Let's deploy the portable serverless function to AWS Lambda, one of the most popular serverless platforms for developers. First, you need to add another `funqy` extension.

Run the following two Maven commands to add the `funqy` extension, `quarkus-funqy-amazon-lambda`, to enable AWS Lambda deployment in Quarkus:

```
$ ./mvnw quarkus:remove-extension
-Dextensions="io.quarkus:quarkus-funqy-
http,quarkus-openshift"

$ ./mvnw quarkus:add-extension -Dextensions="io.
quarkus:quarkus-funqy-amazon-lambda"
```

Note: Remove the `funqy-http` and `quarkus-openshift` extensions since the `quarkus-funqy-amazon-lambda` extension handles HTTP request binding as well.

Update the `hello` method in `src/main/java/GreetingResource.java` to specify a new function name, `awsfunction`, and output:

```
@Funq("awsfunction")
public String hello() {
   return "Hi, Quarkus Funqy on AWS Lambda";
}
```

Comment out all configuration lines in `application.properties`.

Then add the following key and value to export the function name in AWS Lambda:

```
quarkus.funqy.export=awsfunction
```

Building the serverless function using the following Maven command compiles the code and generates all necessary resource files for AWS Lambda deployment and local simulation:

```
$ ./mvnw clean package -DskipTests
```

The following files should be generated in the `target/` directory:

- `function.zip` – AWS Lambda deployment file
- `manage.sh` – Bash script for wrapping AWS CLIs to create and delete AWS Lambda deployment files
- `sam.jvm.yaml` – AWS Serverless Application Model (SAM) CLI script for local testing
- `sam.native.yaml` – SAM CLI script with a native executable file for local testing

Deploy the Quarkus Funqy application to AWS Lambda as a serverless function using the following command:

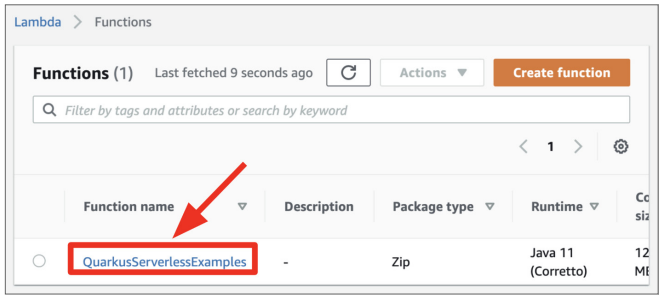
```
$ LAMBDA_ROLE_ARN=<YOUR_ROLE_ARN> sh target/
manage.sh create
```

Note: If you haven't created Amazon Resource Names (ARNs) already, you can learn more here: https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html

The output should end with `"LastUpdateStatus": "Successful"`.

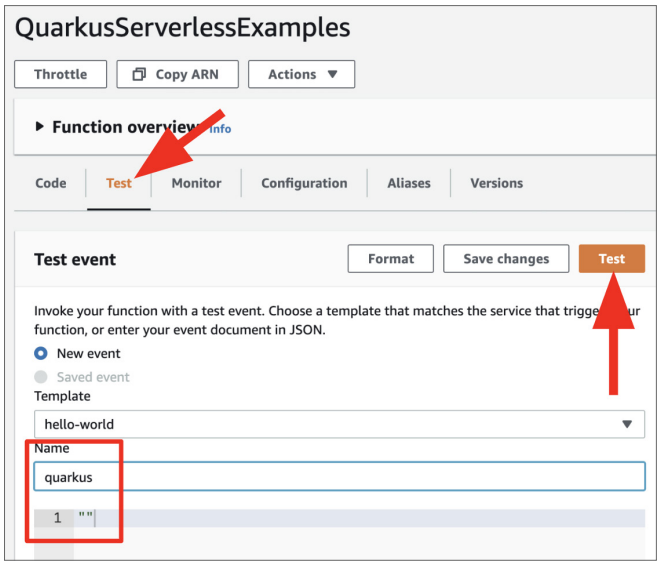
Access the AWS console with your credentials, then navigate to the AWS Lambda service page. You should see that the Quarkus function is already deployed, as shown in Figure 8.

Figure 8: AWS Lambda Service landing page



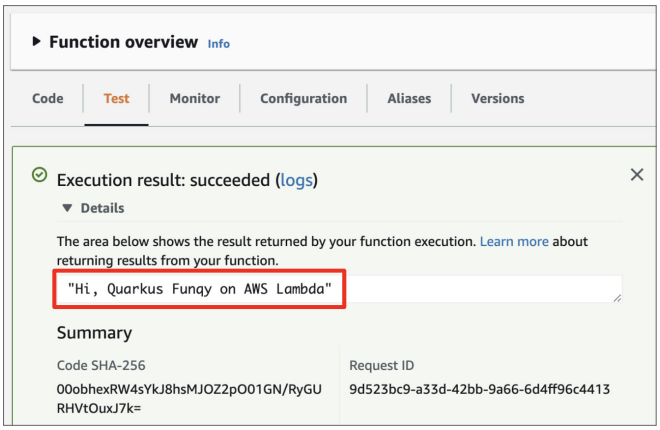
Click on the function's name, `QuarkusServerlessExamples`, to view the *Function Overview* page. Click on the `Test` tab, then input the empty string (e.g., `""`), as shown in Figure 9.

Figure 9: Test the Quarkus function on AWS Lambda



The output should be `"Hi, Quarkus Funqy on AWS Lambda"`. If you wouldn't change the return text, the output should be the same as when you deployed to OpenShift earlier.

Figure 10: Test result on AWS Lambda



You can also delete the function on AWS Lambda quickly using the wrapper script, `manage.sh`, when the function is not required to serve a business feature any longer.

Use the following command in your local environment to delete the function from AWS Lambda:

```
$ LAMBDA_ROLE_ARN=<YOUR_ROLE_ARN> sh target/
manage.sh delete
```

BIND CLOUDEVENTS ON KNATIVE WITH QUARKUS SERVERLESS FUNCTIONS

While events can be used to communicate between services, trigger out-of-band processing, or send a payload to a service like Apache Kafka, developers have to spend more time and effort handling messages in different ways. For example, some messages are payloads in JSON format, while other applications use binary formats such as Avro and Protobuf to transport payloads with metadata.

CloudEvents provides an open specification for describing events with a common format and increases interoperability across multiple cloud platforms and middleware stacks (e.g., Knative, Kogito, Debezium, Quarkus). It also allows developers to decouple binding between event producers and consumers for efficient serverless architectures.

Quarkus `funqy` extensions enable developers to bind CloudEvents for handling reactive streams with serverless functions in a Knative environment. This is beneficial for developers who are building a common messaging format to describe events and increases interoperability among multi- and hybrid cloud platforms.

Remove the `funqy-amazon-lambda` extension and add the `quarkus-funqy-knative-events` and `quarkus-openshift` extensions to bind a CloudEvent on Knative with Quarkus serverless functions:

```
$ ./mvnw quarkus:remove-extension
-Dextensions="io.quarkus:quarkus-funqy-amazon-
lambda"

$ ./mvnw quarkus:add-extension -Dextensions="io.
quarkus:quarkus-funqy-knative-events,quarkus-
openshift"
```

Add a new function (e.g., `ToLowercaseFunction`) to process the CloudEvent messages in the Quarkus project, as shown in the example below:

```
public class ToLowercaseFunction {

   @Funq("lowercase")
   public Output function(Input input, @Context
CloudEvent<Input> cloudEvent) {
      String inputStr = input.getInput();
      String outputStr = Optional.
ofNullable(inputStr)
         .map(String::toLowerCase)
         .orElse("NO DATA");

      LOGGER.info("Output CE: {}", outputStr);
      return new Output(inputStr, cloudEvent.
subject(), outputStr, null);
   }
}
```

CODE CONTINUES IN NEXT COLUMN

Note: `Input` and `Output` classes are required to invoke the `function` method on your local filesystem. Find the classes in the GitHub repository.

Uncomment all configuration lines in `application.properties` and replace the function name with `lowercase`:

```
quarkus.funqy.export=lowercase
```

Rebuild and redeploy the function to OpenShift using the following Maven command:

```
$ ./mvnw clean package -DskipTests
```

Send a CloudEvent message to the serverless function in OpenShift:

```
$ URL=http://quarkus-serverless-[...].SUBDOMAIN
$ curl -v ${URL} \
   -H "Content-Type:application/json" \
   -H "Ce-Id:1" \
   -H "Ce-Source:quarkus-cloudevent-example" \
   -H "Ce-Type:dzone.refcard.quarkus" \
   -H "Ce-1.0" \
   -d "{\"input\": \"QUARKUS WITH CLOUDEVENT\"}\""
```

Again, access the running Quarkus serverless pod's logs in OpenShift, and you will see the result of processing the CloudEvent (e.g., `input` and `output`):

```
...
INFO  [org.acm.get.sta.ToLowercaseFunction]
(executor-thread-0) Input: Input{input='QUARKUS
WITH CLOUDEVENT'}
INFO  [org.acm.get.sta.ToLowercaseFunction]
(executor-thread-0) Output CE: quarkus with
cloudevent
```

CONCLUSION

A serverless development model is now required for enterprises that want to spin up their business applications on demand rather than run them all the time. Many enterprises are considering new programming languages other than Java to maximize serverless effectiveness due to new cloud-native deployment models with containers and Kubernetes.

This Refcard demonstrated how to get started with Quarkus for serverless function development using key features like Quarkus' built-in live coding, continuous testing, rich graphical DEV UI, and DevServices. Continue your Quarkus journey here!

ADDITIONAL RESOURCES

For more in-depth coverage of the sections in this Refcard, you can read the following articles on dzone.com and other helpful resources:

- "What Is Serverless With Java?" https://dzone.com/articles/what-is-serverless-with-java
- "Getting Started With Java Serverless Functions Using Quarkus and AWS Lambda" https://dzone.com/articles/getting-started-with-java-serverless-functions-usi
- "Get Started With Java Serverless Functions" https://dzone.com/articles/get-started-with-java-serverless-functions
- "Optimize Java Serverless Functions in Kubernetes" https://dzone.com/articles/optimize-java-serverless-functions-in-kubernetes
- "Making portable functions across serverless platforms" https://dzone.com/articles/making-portable-functions-across-serverless-platfo
- "Bind a Cloud Event to Knative" https://dzone.com/articles/bind-a-cloud-event-to-knative
- "A Guide to Java Serverless Functions eBook" https://opensource.com/downloads/java-serverless-ebook
- "Getting Started With Quarkus" Refcard https://dzone.com/refcardz/quarkus-1

WRITTEN BY DANIEL OH,
SR. PRINCIPAL TECHNICAL MARKETING MANAGER, RED HAT

Daniel Oh is a well-known public speaker, open-source contributor, published author, and developer advocate for 20+ years of experience in solving real-world enterprise problems using cloud-native runtimes (e.g., Quarkus, Spring Boot) on Kubernetes. He's also a Cloud Native Computing Foundation Ambassador for evangelizing DevOps teams for developing cloud-native microservices, serverless functions, then deploying them to hybrid clouds in easy-to-use and cost-effective ways.

DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709

888.678.0399 | 919.678.0300