A guide to Java serverless functions

Why choose Java for serverless application development and how to get started

ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com

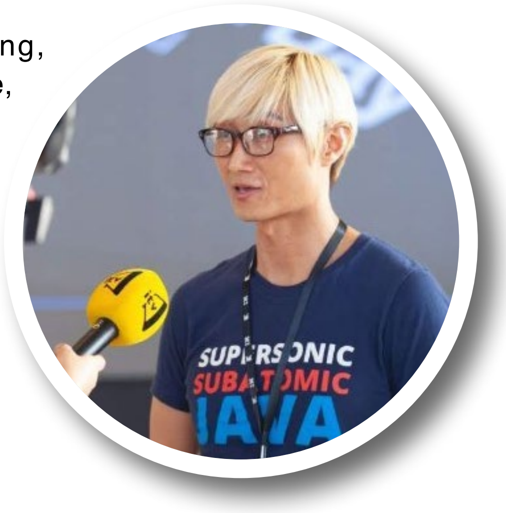
open source .com

Supported by Red Hat

DANIEL OH

DANIEL OH Technical Marketing, Developer Advocate, CNCF Ambassador, Public Speaker, Published Author, Quarkus, Red Hat Runtimes.

Follow me at @danieloh30



CONTENTS

What is serverless with Java?

Java is still one of the most popular languages for developing enterprise applications. So, why are serverless developers shying away from it?

FOR DECADES, ENTERPRISES have developed business-critical applications on various platforms, including physical servers, virtual machines, and cloud environments. The one thing these applications have in common across industries is they need to be continuously available (24x7x365) to guarantee stability, reliability, and performance, regardless of demand. Therefore, every enterprise must be responsible for the high costs of maintaining an infrastructure (e.g., CPU, memory, disk, networking, etc.) even if actual resource utilization is less than 50%.

Serverless architecture was developed to help solve these problems. Serverless allows developers to build and run applications on demand, guaranteeing high availability without having to manage servers in multi- and hybrid-cloud environments. Behind the scenes, there are still many servers in the serverless topology, but they are abstracted away from application development. Instead, cloud providers use serverless services for resource management, such as provisioning, maintaining, networking, and scaling server instances.

Because of its effectiveness, the serverless development model is now a requirement for enterprises that want to spin up their business applications on demand rather than running them all the time.

Many open source projects have been created to manage serverless applications on Kubernetes [1] clusters with the Linux container package over virtual machines. The CNCF's Interactive Serverless Landscape [2] is a guide to open source projects, tools, frameworks, and public cloud platforms that enable DevOps teams to handle serverless applications.

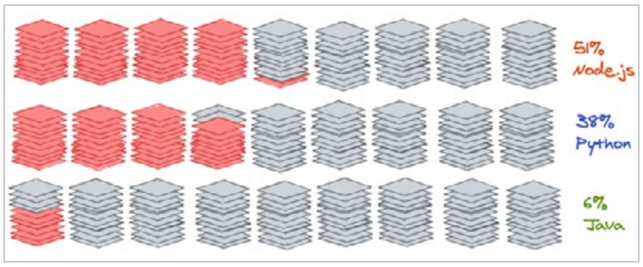
Tools
Security
Framework
dapr
Hosted Platform
Installable Platform

(*CNCF, Apache License 2.0*)

Developers can write code then deploy it quickly to various serverless environments. Then the serverless application responds to demand and automatically scales up and down as needed.

You may be wondering what programming language and runtime are best suited for serverless application development to integrate with the technologies in the figure above. There's not just one answer to this question, but let's take a step back to discuss the application runtime that is most popular for developing business applications in enterprise production environments: Java.

According to Developer Economics [3], as of Q3 2020, more than 8 million enterprise developers are still using Java to achieve their business requirements. Yet, according to a 2020 NewRelic survey [4], Java (at 6%) is clearly not the top choice for forward-thinking developers using a popular cloud service.



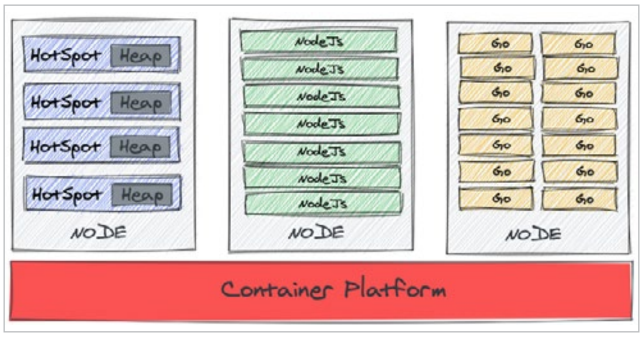
Data from NewRelic's Serverless Benchmark Report (Daniel Oh, CC BY-SA 4.0)

Resource usage, response times, and latency are critical in serverless development. Serverless offerings from public cloud providers are usually metered on-demand, charged only when a serverless application is up, through an event-driven execution model. Therefore, enterprises don't pay anything when a serverless application is idle or scaled down to zero.

The state of Java with containers

With this background, you may be asking: "*Why don't developers try to use the Java stack for serverless application development given that existing business applications are most likely developed on Java technologies?*"

Here is the hidden truth: It's hard to optimize Java applications in the new immutable infrastructure, also known as container platforms (e.g., Kubernetes).



(Daniel Oh, CC BY-SA 4.0)

This diagram depicts the differences in memory resource usage between a Java process and competing languages and frameworks, such as Node.js [5] and Go [6]. Java HotSpot has the largest footprint, which includes the heap memory allocated per Java Virtual Machine (JVM) instance. The middle shows how much smaller each process is on Node.js compared to Java. And finally, Go is a compiled language popular on the cloud due to its low memory consumption.

As you can see, you get higher density as you go from left to right on this diagram. This is the reason developers shy away from Java (including Spring Boot [7], an opinionated microservice Java framework) when writing serverless applications on the cloud, containers, and Kubernetes.

What's next?

Enterprises can gain significant benefits by implementing serverless applications, but resource-density issues cause them to avoid using the Java stack for developing serverless application development on Kubernetes. But choosing a different language creates a burden on the millions of Java developers worldwide. Therefore, in the next article in this series, I will guide you on how to get started with Java serverless functions instead of choosing a different language.

Links

[1] https://opensource.com/article/19/6/reasons-kubernetes
[2] https://landscape.cncf.io/serverless?zoom=150
[3] https://developereconomics.com/
[4] https://newrelic.com/resources/ebooks/serverless-benchmark-report-aws-lambda-2020
[5] https://nodejs.org/
[6] https://golang.org/
[7] https://spring.io/projects/spring-boot

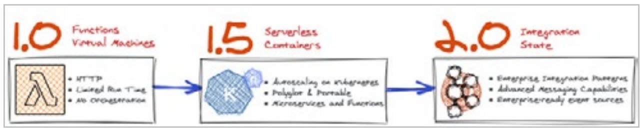
Write your own **serverless functions**

Quarkus allows you to develop serverless workloads with familiar Java technology.

THE SERVERLESS JAVA JOURNEY [1] started out with functions—small snippets of code running on demand. This phase didn't last long. Although functions based on virtual machine architecture in the 1.0 phase made this paradigm very popular, as the graphic below shows, there were limits around execution time, protocols, and poor local-development experience.

Developers then realized that they could apply the same serverless traits and benefits to microservices and Linux containers. This launched the 1.5 phase, where some serverless containers completely abstracted Kubernetes [2], delivering the serverless experience through Knative [3] or another abstraction layer that sits on top of it.

In the 2.0 phase, serverless starts to handle more complex orchestration and integration patterns combined with some level of state management. More importantly, developers want to keep using a familiar application runtime, Java, to run a combination of serverless and non-serverless workloads in legacy systems.


1.0
Functions
Virtual Machines
• HTTP
• Limited Run Time
• No Orchestration
1.5
Serverless
Containers
• Autoscaling on Kubernetes
• Polyglot & Portable
• Microservices and Functions
2.0
Integration
State
• Enterprise Integration Patterns
• Advanced Messaging Capabilities
• Enterprise-ready event sources

(Daniel Oh, CC BY-SA 4.0)

Before Java developers can start developing new serverless functions, their first task is to choose a new cloud-native Java framework that allows them to run Java functions quicker with a smaller memory footprint than traditional monolithic applications. This can be applied to various infrastructure environments, from physical servers to virtual machines to containers in multi- and hybrid-cloud environments.

Developers might consider an opinionated Spring framework that uses the `java.util.function` package in Spring Cloud Function [4] to support the development of imperative and reactive functions. Spring also enables developers to deploy Java functions to installable serverless platforms such as Kubeless [5], Apache OpenWhisk [6], Fission [7], and Project Riff [8]. However, there are concerns about slow startup and response times and heavy memory-consuming processes with Spring. This problem can be worse when running Java functions on scalable container environments such as Kubernetes.

Quarkus [9] is a new open source cloud-native Java framework that can help solve these problems. It aims to design serverless applications and write cloud-native microservices for running on cloud infrastructures (e.g., Kubernetes).

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. Quarkus also includes more than 100 extensions that integrate enterprise capabilities, including database access, serverless integration, messaging, security, observability, and business automation.

Here is a quick example of how developers can scaffold a Java serverless function project with Quarkus.

1. Create a Quarkus serverless Maven project

Developers have multiple options to install a local Kubernetes cluster, including Minikube [10] and OKD (OpenShift Kubernetes Distribution) [11]. This tutorial uses an OKD cluster for a developer's local environment because of the easy setup of serverless functionality on Knative and DevOps toolings. These guides for OKD installation [12] and Knative operator installation [13] offer more information about setting them up.

The following command generates a Quarkus project (e.g., `quarkus-serverless-restapi`) to expose a simple REST API and download a `quarkus-openshift` extension for Knative service deployment:

```
$ mvn io.quarkus:quarkus-maven-plugin:2.0.3.Final:create \
       -DprojectGroupId=org.acme \
       -DprojectArtifactId=quarkus-serverless-restapi \
       -Dextensions="openshift" \
       -DclassName="org.acme.getting.started.GreetingResource"
```

2. Run serverless functions locally

Run the application using Quarkus development mode to check if the REST API works, then tweak the code a bit:

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

The output will look like this:

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (Quarkus Main Thread) quarkus-serverless-
  restapi 1.0.0-SNAPSHOT on JVM (powered by Quarkus xx.xx.xx.)
  started in 2.386s. Listening on: http://localhost:8080
INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated.
  Live Coding activated.
INFO  [io.quarkus] (Quarkus Main Thread) Installed features:
  [cdi, kubernetes, resteasy]
```

Note: Keep your Quarkus application running to use Live Coding. This allows you to avoid having to rebuild, redeploy the application, and restart the runtime whenever the code changes.

Now you can hit the REST API with a quick `curl` command. The output should be Hello `RESTEasy`:

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

Tweak the return text in `GreetingResource.java`:

```
    public String hello() {
        return "Quarkus Function on Kubernetes";
    }
```

You will see new output when you reinvoke the REST API:

```
$ curl localhost:8080/hello
Quarkus Function on Kubernetes
```

There's not been a big difference between normal microservices and serverless functions. A benefit of Quarkus is that it enables developers to use any microservice to deploy Kubernetes as a serverless function.

3. Deploy the functions to a Knative service

If you haven't already, create a namespace [14] (e.g., `quarkus-serverless-restapi`) on your OKD (Kubernetes) cluster to deploy this Java serverless function.

Quarkus enables developers to generate Knative and Kubernetes resources by adding the following variables in `src/main/resources/application.properties`:

```
quarkus.container-image.group=quarkus-serverless-restapi <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>
```

<1> Define a project name where you deploy a serverless application
<2> The container registry to use
<3> Use self-signed certs in this simple example to trust them
<4> Enable the generation of Knative resources
<5> Instruct the extension to deploy to OpenShift after the container image is built
<6> Set the Docker build strategy

This command builds the application then deploys it directly to the OKD cluster:

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

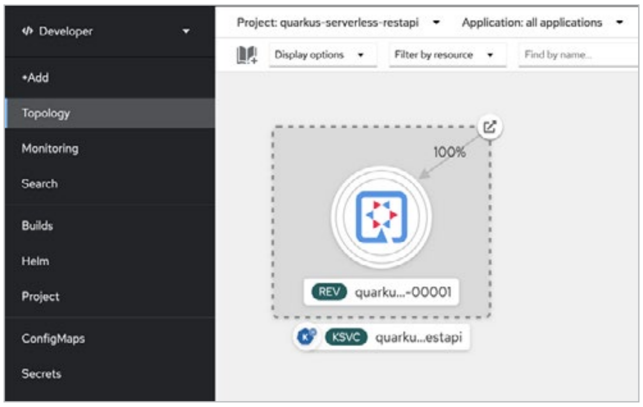
Note: Make sure to log in to the right project (e.g., quarkus-serverless-restapi) by using the oc login command ahead of time.

The output should end with `BUILD SUCCESS`.

Add a Quarkus label to the Knative service with this `oc` command:

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

Then access the OKD web console to go to the Topology view in the Developer perspective [15]. You might see that your pod (serverless function) is already scaled down to zero (white-line circle).



(Daniel Oh, CC BY-SA 4.0)

4. Test the functions on Kubernetes

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

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

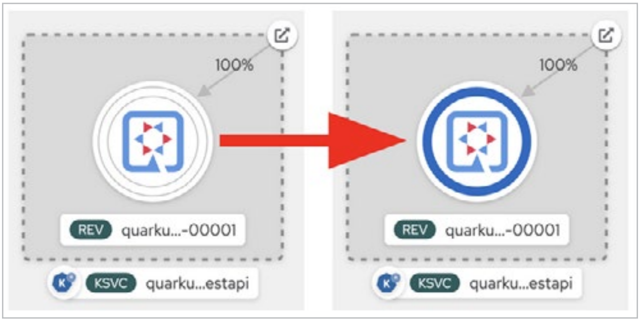
Access the route URL with a curl command:

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

In a few seconds, you will get the same result as you got locally:

```
Quarkus Function on Kubernetes
```

When you return to the Topology view in the OKD cluster, the Knative service scales up automatically.



(Daniel Oh, CC BY-SA 4.0)

This Knative service pod will go down to zero again in 30 seconds because of Knative serving's default setting.

What's next?

The serverless journey has evolved, starting with functions on virtual machines to serverless containers and integration with enterprise legacy systems. Along this journey, enterprise developers can still use familiar technologies like Java for developing serverless functions by using Quarkus to create a project then build and deploy it to Kubernetes with a Knative service.

The next article in this series will guide you on optimizing Java serverless functions in Kubernetes for faster startup time and small memory footprints at scale.

Links

[1] https://opensource.com/article/21/5/what-serverless-java
[2] https://opensource.com/article/19/6/reasons-kubernetes
[3] https://cloud.google.com/knative/
[4] https://spring.io/serverless
[5] https://kubeless.io/
[6] https://openwhisk.apache.org/
[7] https://fission.io/
[8] https://projectriff.io/
[9] https://quarkus.io/
[10] https://minikube.sigs.k8s.io/docs/start/
[11] https://docs.okd.io/latest/welcome/index.html
[12] https://docs.okd.io/latest/installing/index.html
[13] https://knative.dev/docs/install/knative-with-operators/
[14] https://docs.okd.io/latest/applications/projects/configuring-project-creation.html
[15] https://docs.okd.io/latest/applications/application_life_cycle_management/odc-viewing-application-composition-using-topology-view.html

Make your app run faster with GraalVM in Kubernetes

Achieve faster startup and a smaller memory footprint to run serverless functions on Kubernetes.

A FASTER STARTUP and smaller memory footprint always matter in Kubernetes [1] due to the expense of running thousands of application pods and the cost savings of doing it with fewer worker nodes and other resources. Memory is more important than throughput on containerized microservices on Kubernetes because:

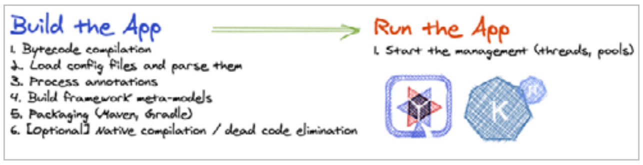
- It's more expensive due to permanence (unlike CPU cycles)
- Microservices multiply the overhead cost
- One monolith application becomes *N* microservices (e.g., 20 microservices ≈ 20GB)

This significantly impacts serverless function development and the Java deployment model. This is because many enterprise developers chose alternatives such as Go, Python, and Nodejs to overcome the performance bottleneck—until now, thanks to Quarkus [2], a new Kubernetes-native Java stack. This article explains how to optimize Java performance to run serverless functions on Kubernetes using Quarkus.

Container-first design

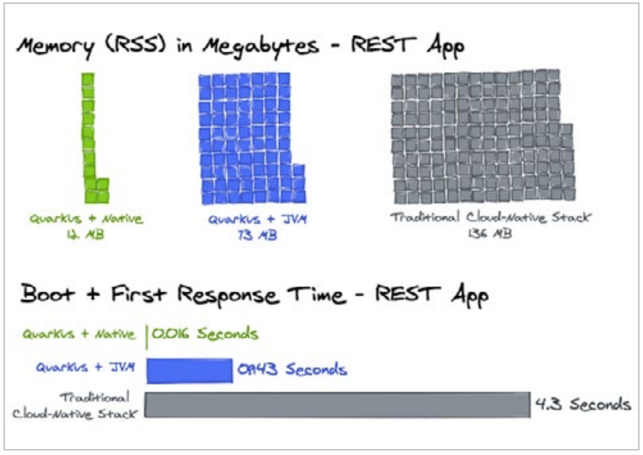
Traditional frameworks in the Java ecosystem come at a cost in terms of the memory and startup time required to initialize those frameworks, including configuration processing, classpath scanning, class loading, annotation processing, and building a metamodel of the world, which the framework requires to operate. This is multiplied over and over for different frameworks.

Quarkus helps fix these Java performance issues by "shifting left" almost all of the overhead to the build phase. By doing code and framework analysis, bytecode transformation, and dynamic metamodel generation only once, at build time, you end up with a highly optimized runtime executable that starts up super fast and doesn't require all the memory of a traditional startup because the work is done once, in the build phase.


Build the App
1. Bytecode compilation
2. Load config files and parse them
3. Process annotations
4. Build framework meta-models
5. Packaging (Maven, Gradle)
6. [Optional] Native compilation / dead code elimination
Run the App
1. Start the management (threads, pools)

(Daniel Oh, CC BY-SA 4.0)

More importantly, Quarkus allows you to build a native executable file that provides performance advantages [3], including amazingly fast boot time and incredibly small resident set size (RSS) memory, for instant scale-up and high-density memory utilization compared to the traditional cloud-native Java stack.



(Daniel Oh, CC BY-SA 4.0)

Here is a quick example of how you can build the native executable with a Java serverless [4] function project using Quarkus.

1. Create the Quarkus serverless Maven project

This command generates a Quarkus project (e.g., `quarkus-serverless-native`) to create a simple function:

```
$ mvn io.quarkus:quarkus-maven-plugin:2.0.3.Final:create \
       -DprojectGroupId=org.acme \
       -DprojectArtifactId=quarkus-serverless-native \
       -DclassName="org.acme.getting.started.GreetingResource"
```

2. Build a native executable

You need a GraalVM to build a native executable for the Java application. You can choose any GraalVM distribution, such as Oracle GraalVM Community Edition (CE) [5] and Mandrel [6] (the downstream distribution of Oracle GraalVM CE). Mandrel is designed to support building Quarkus-native executables on OpenJDK 11.

Open `pom.xml`, and you will find this native profile. You'll use it to build a native executable:

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

> **Note:** You can install the GraalVM or Mandrel distribution locally. You can also download the Mandrel container image to build it (as I did), so you need to run a container engine (e.g., Docker) locally.

Assuming you have started your container runtime already, run one of the following Maven commands.

For Docker [7]:

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

For Podman [8]:

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

The output should end with `BUILD SUCCESS`.

```
INFO  [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] Running Quarkus native-image plugin on GraalVM Version [illegible]
INFO  [io.quarkus.deployment.pkg.steps.NativeImageBuildRunner] docker run [illegible]
[illegible]
WARNING [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] objcopy executable not found in PATH. Debug symbols will not be separated from executable.
WARNING [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] That will result in a larger native image with debug symbols embedded in it.
INFO  Checking for existing resources in: [illegible]/quarkus-serverless-native/src/main/kubernetes.
INFO  [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in [illegible]ms
INFO  ------------------------------------------------------------------------
INFO  BUILD SUCCESS
INFO  ------------------------------------------------------------------------
INFO  Total time:  [illegible] min
INFO  Finished at: [illegible]
```

(Daniel Oh, CC BY-SA 4.0)

Run the native executable directly without Java Virtual Machine (JVM):

```
$ target/quarkus-serverless-native-1.0.0-SNAPSHOT-runner
```

The output will look like:

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (main) quarkus-serverless-native
1.0.0-SNAPSHOT native
(powered by Quarkus xx.xx.xx.) Started in 0.019s. Listening on:
   http://0.0.0.0:8080
INFO [io.quarkus] (main) Profile prod activated.
INFO [io.quarkus] (main) Installed features: [cdi, kubernetes,
   resteasy]
```

Supersonic! That's *19 milliseconds* to startup. The time might be different in your environment.

It also has extremely low memory usage, as the Linux `ps` utility reports. While the app is running, run this command in another terminal:

```
$ ps -o pid,rss,command -p $(pgrep -f runner)
```

You should see something like:

```
  PID    RSS COMMAND
10246  11360 target/quarkus-serverless-native-1.0.0-SNAPSHOT-runner
```

This process is using around *11MB* of memory (RSS). Pretty compact!

> **Note:** The RSS and memory usage of any app, including Quarkus, will vary depending on your specific environment and will rise as application experiences load.

You can also access the function with a REST API. Then the output should be `Hello RESTEasy`:

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

3. Deploy the functions to Knative service

If you haven't already, create a namespace [9] (e.g., `quarkus-serverless-native`) on OKD (OpenShift Kubernetes Distribution) [10] to deploy this native executable as a serverless function. Then add a `quarkus-openshift` extension for Knative service deployment:

```
$ ./mvnw -q quarkus:add-extension -Dextensions="openshift"
```

Append the following variables in `src/main/resources/application.properties` to configure Knative and Kubernetes resources:

```
quarkus.container-image.group=quarkus-serverless-native
quarkus.container-image.registry=image-registry.openshift-
  image-registry.svc:5000
quarkus.native.container-build=true
quarkus.kubernetes-client.trust-certs=true
quarkus.kubernetes.deployment-target=knative
quarkus.kubernetes.deploy=true
quarkus.openshift.build-strategy=docker
```

Build the native executable, then deploy it to the OKD cluster directly:

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

> **Note:** Make sure to log in to the right project (e.g., `quarkus-serverless-native`) using the `oc login` command ahead of time.

The output should end with `BUILD SUCCESS`. It will take a few minutes to complete a native binary build and deploy a new Knative service. After successfully creating the service, you should see a Knative service (KSVC) and revision (REV) using either the `kubectl` or `oc` command tool:

```
$ kubectl get ksvc
NAME                        URL   [...]
quarkus-serverless-native   http://quarkus-serverless-native-[...].SUBDOMAIN  True

$ kubectl get rev
NAME                              CONFIG NAME                 K8S SERVICE NAME                  GENERATION  READY REASON
quarkus-serverless-native-00001   quarkus-serverless-native   quarkus-serverless-native-00001   1           True
```

4. Access the native executable function

Retrieve the serverless function's endpoint by running this `kubectl` command:

```
$ kubectl get rt/quarkus-serverless-native
```

The output should look like:

```
NAME                      URL                                                                   READY   REASON
quarkus-serverless-native http://quarkus-serverless-restapi-quarkus-serverless-native.SUBDOMAIN   True
```

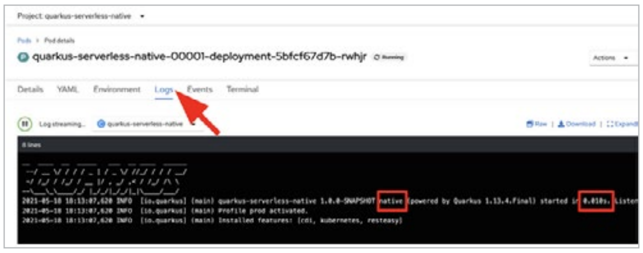
Access the `route URL` with a `curl` command:

```
$ curl http://quarkus-serverless-restapi-quarkus-serverless-
  native.SUBDOMAIN/hello
```

In less than one second, you will get the same result as you got locally:

```
Hello RESTEasy
```

When you access the Quarkus running pod's logs in the OKD cluster, you will see the native executable is running as the Knative service.


Project: quarkus-serverless-native
Pods > Pod details
quarkus-serverless-native-00001-deployment-5bfcf67d7b-rwhjr
Running
Actions
Details
YAML
Environment
Logs
Events
Terminal
Log streaming...
quarkus-serverless-native
Raw
Download
Expand
8 lines

```
2021-05-18 18:13:07,620 INFO  [io.quarkus] (main) quarkus-serverless-native 1.0.0-SNAPSHOT native (powered by Quarkus 1.13.4.Final) started in 0.010s. Listen
2021-05-18 18:13:07,620 INFO  [io.quarkus] (main) Profile prod activated.
2021-05-18 18:13:07,620 INFO  [io.quarkus] (main) Installed features: [cdi, kubernetes, resteasy]
```

(Daniel Oh, CC BY-SA 4.0)

What's next?

You can optimize Java serverless functions with GraalVM distributions to deploy them as serverless functions on Knative with Kubernetes. Quarkus enables this performance optimization using simple configurations in normal microservices.

The next article in this series will guide you on making portable functions across multiple serverless platforms with no code changes.

Links

[1] https://opensource.com/article/19/6/reasons-kubernetes
[2] https://quarkus.io/
[3] https://quarkus.io/blog/runtime-performance/
[4] https://opensource.com/article/21/5/what-serverless-java
[5] https://www.graalvm.org/community/
[6] https://github.com/graalvm/mandrel
[7] https://www.docker.com/
[8] https://podman.io/
[9] https://docs.okd.io/latest/applications/projects/configuring-project-creation.html
[10] https://docs.okd.io/latest/welcome/index.html

Making portable functions across serverless platforms

Quarkus Funqy brings portability to serverless functions.

THE RISING POPULARITY of serverless development alongside the increased adoption of multi- and hybrid-cloud architectures has created a lot of competition among platforms. This gives developers many choices about where they can run functions on serverless platforms—from public managed services to on-premises Kubernetes [1].

If you've read my previous articles about Java serverless [2], you learned how to get started developing Java serverless functions [3] with Quarkus and how those serverless functions can be optimized [4] to run on Kubernetes. So what should you do next to make your serverless functions fit better with the many choices available to you?

As a clue, think about why the Linux container (Docker, LXC [5], cri-o) has become so popular: Portability. It's what made containers the de facto packaging technology for moving things from a developer's local machine to Kubernetes environments at scale. It means developers and operators don't need to worry about incompatibility and inconsistency between development and production environments.

For adopting multi- and hybrid cloud architectures, these container portability benefits should also be considered for serverless function development. Without portability, developers would likely have to learn and use different APIs, command-line interface (CLI) tools, and software development kits (SDKs) for each serverless platform when developing and deploying the same serverless functions across multiple serverless runtimes. Developers, who have limited resources (e.g., time, effort, cost, and human resources), would be so overwhelmed by the options that they would find it difficult to choose the best one.

Tools
Framework
Hosted Platform
Installable Platform
Security
Too many choices

(Daniel Oh, CC BY-SA 4.0)

Get Funqy the next time you hit a serverless dance floor

The Quarkus Funqy [6] extension supports a portable Java API for developers to write serverless functions and deploy them to heterogeneous serverless runtimes, including AWS Lambda, Azure Functions, Google Cloud, and Knative. It is also usable as a standalone service. Funqy helps developers dance on the serverless floor without making code changes.

Here is a quick example of how to build a portable serverless function with Quarkus Funqy.

1. Create a Quarkus Funqy Maven project

Generate a Quarkus project (`quarkus-serverless-func`) to create a simple function with Funqy extensions:

```
$ mvn io.quarkus:quarkus-maven-plugin:2.0.3.Final:create \
       -DprojectGroupId=org.acme \
       -DprojectArtifactId=quarkus-serverless-func \
       -Dextensions="funqy-http" \
       -DclassName="org.acme.getting.started.GreetingResource"
```

2. Run the serverless function locally

Open the `Funqy.java` file in the `src/main/java/org/acme/getting/started` directory:

```
public class Funqy {

    private static final String CHARM_QUARK_SYMBOL = "c";

    @Funq (1)
    public String charm(Answer answer) { (2)
        return CHARM_QUARK_SYMBOL.equalsIgnoreCase(answer.
          value) ? "You Quark!" : "👻Wrong answer";
    }

    public static class Answer {
        public String value; (3)
    }
}
```

In the code above:

(1) Annotation makes the method an exposable function based on the Funqy API. The function name is equivalent to the method name (`charm`) by default.
(2) Indicates a Java class (`Answer`) as an input parameter and `String` type for the output.
(3) `value` should be parameterized when the function is invoked.

> **Note:** Funqy does type introspection at build time to speed boot time, so the Funqy marshaling layer won't notice any derived types at runtime.

Run the function via Quarkus Dev Mode:

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

The output should look like:

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (Quarkus Main Thread) quarkus-serverless-
  func 1.0.0-SNAPSHOT on JVM (powered by Quarkus x.x.x.)
  started in 2.908s. Listening on: http://localhost:8080
INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated.
  Live Coding activated.
INFO  [io.quarkus] (Quarkus Main Thread) Installed features:
  [cdi, funqy-http, kubernetes]
```

Now the function is running in your local development environment. Access the function with a RESTful API:

```
$ http://localhost:8080/charm?value=s
```

The output should be:

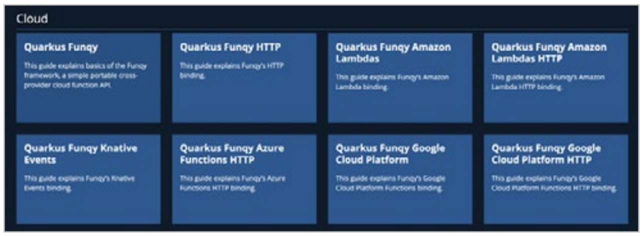
```
👻 Wrong answer
```

If you pass `value=c` down as a parameter, you will see:

```
You Quark!
```

3. Choose a serverless platform to deploy the Funqy function

Now you can deploy the portable function to your preferred serverless platform when you add one of the Quarkus Funqy extensions in the figure below. The advantage is that you will not need to change the code; you should need only to adjust a few configurations, such as function export and target serverless platform.



Cloud

Quarkus Funqy
This guide explains basics of the Funqy framework, a simple portable cross-provider cloud function API.

Quarkus Funqy HTTP
This guide explains Funqy's HTTP binding.

Quarkus Funqy Amazon Lambdas
This guide explains Funqy's Amazon Lambda binding.

Quarkus Funqy Amazon Lambdas HTTP
This guide explains Funqy's Amazon Lambda HTTP binding.

Quarkus Funqy Knative Events
This guide explains Funqy's Knative Events binding.

Quarkus Funqy Azure Functions HTTP
This guide explains Funqy's Azure Functions HTTP binding.

Quarkus Funqy Google Cloud Platform
This guide explains Funqy's Google Cloud Platform Functions binding.

Quarkus Funqy Google Cloud Platform HTTP
This guide explains Funqy's Google Cloud Platform Functions HTTP binding.

(Daniel Oh, CC BY-SA 4.0)

Try to deploy the function using Knative Serving [7] (if you have installed it in your Kubernetes cluster). Add the following extensions to the Quarkus Funqy project:

```
$ ./mvnw quarkus:add-extension -Dextensions=
  "kubernetes,container-image-docker"
```

Open the `application.properties` file in the `src/main/resources/` directory. Then add the following variables to configure Knative and Kubernetes resources—make sure to replace changeit with your container registry's group name (username in DockerHub):

```
quarkus.container-image.build=true
quarkus.container-image.group=changeit
quarkus.container-image.push=true
quarkus.container-image.builder=docker
quarkus.kubernetes.deployment-target=knative
```

Containerize the function, then push it to the external container registry:

```
$ ./mvnw clean package
```

The output should end with `BUILD SUCCESS`. Then a `knative.yml` file will be generated in the `target/kubernetes` directory. Now you should be ready to create a Knative service with the function using the following command (be sure to log into the Kubernetes cluster and change the namespace where you want to create the Knative service):

```
$ kubectl create -f target/kubernetes/knative.yml
```

The output should be like this:

```
service.serving.knative.dev/quarkus-serverless-func created
```

4. Test the Funqy function in Kubernetes

Get the function's REST API and note its output:

```
$ kubectl get rt
NAME URL READY REASON
quarkus-serverless-func
  http://quarkus-serverless-func-YOUR_HOST_DOMAIN   True
```

Access the function quickly using a curl command:

```
$ http://http://quarkus-serverless-func-YOUR_HOST_DOMAIN/
  charm?value=c
```

You see the same output as you saw locally:

```
You Quark!
```

Note: The function will scale down to zero in 30 seconds because of Knative Serving's default behavior. In this case, the pod will scale up automatically when the REST API is invoked.

What's next?

You've learned how developers can make portable Java serverless functions with Quarkus and deploy them across serverless platforms (e.g., Knative with Kubernetes). Quarkus enables developers to avoid redundancy when creating the same function and deploying it to multiple serverless platforms. My next article in this series will explain how to enable CloudEvents Bind with Java and Knative.

Links

[1] https://opensource.com/article/19/6/reasons-kubernetes
[2] https://opensource.com/article/21/5/what-serverless-java
[3] https://opensource.com/article/21/6/java-serverless-functions
[4] https://opensource.com/article/21/6/java-serverless-functions-kubernetes
[5] https://www.redhat.com/sysadmin/exploring-containers-lxc
[6] https://quarkus.io/guides/funqy
[7] https://knative.dev/docs/serving/

Bind a cloud event to Knative

CloudEvents provides a common format to describe events and increase interoperability.

EVENTS HAVE BECOME an essential piece of modern reactive systems. Indeed, events can be used to communicate from one service to another, trigger out-of-band processing, or send a payload to a service like Kafka. The problem is that event publishers may express event messages in any number of different ways, regardless of content. For example, some messages are payloads in JSON format to serialize and deserialize messages by application. Other applications use binary formats such as Avro [1] and Protobuf [2] to transport payloads with metadata. This is an issue when building an event-driven architecture that aims to easily integrate external systems and reduce the complexity of message transmission.

CloudEvents [3] is an open specification providing a common format to describe events and increase interoperability. Many cloud providers and middleware stacks, including Knative [4], Kogito [5], Debezium [6], and Quarkus [7] have adopted this format after the release of CloudEvents 1.0. Furthermore, developers need to decouple relationships between event producers and consumers in serverless architectures. Knative Eventing [8] is consistent with the CloudEvents specification, providing common formats for creating, parsing, sending, and receiving events in any programming language. Knative Eventing also enables developers to late-bind event sources and event consumers. For example, a cloud event using JSON might look like this:

```
{
    "specversion" : "1.0", (1)
    "id" : "11111", (2)
    "source" : "http://localhost:8080/cloudevents", (3)
    "type" : "knative-events-binding", (4)
    "subject" : "cloudevents", (5)
    "time" : "2021-06-04T16:00:00Z", (6)
    "datacontenttype" : "application/json", (7)
    "data" : "{\"message\": \"Knative Events\"}", (8)
}
```

In the above code:
(1) Which version of the CloudEvents specification to use
(2) The ID field for a specific event; combining the `id` and the `source` provides a unique identifier
(3) The Uniform Resource Identifier (URI) identifies the event source in terms of the context where it happened or the application that emitted it
(4) The type of event with any random words
(5) Additional details about the event (optional)
(6) The event creation time (optional)
(7) The content type of the data attribute (optional)
(8) The business data for the specific event

Here is a quick example of how developers can enable a CloudEvents bind with Knative and the Quarkus Funqy extension [9].

1. Create a Quarkus Knative event Maven project

Generate a Quarkus project (e.g., `quarkus-serverless-cloudevent`) to create a simple function with Funqy Knative events binding extensions:

```
$ mvn io.quarkus:quarkus-maven-plugin:2.0.3.Final:create \
       -DprojectGroupId=org.acme \
       -DprojectArtifactId=quarkus-serverless-cloudevent \
       -Dextensions="funqy-knative-events" \
       -DclassName="org.acme.getting.started.GreetingResource"
```

2. Run the serverless event function locally

Open the `CloudEventGreeting.java` file in the `src/main/java/org/acme/getting/started/funqy/cloudevent` directory. The `@funq` annotation enables the `myCloudEventGreeting` method to map the input data to the cloud event message automatically:

```
private static final Logger log =
  Logger.getLogger(CloudEventGreeting.class);

    @Funq
    public void myCloudEventGreeting(Person input) {
        log.info("Hello " + input.getName());
    }
}
```

Run the function via Quarkus Dev Mode:

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

The output should look like this:

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
INFO  [io.quarkus] (Quarkus Main Thread) quarkus-serverless-
  cloudevent 1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.0.0.CR3)
  started in 1.546s. Listening on: http://localhost:8080
INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated.
```

```
  Live Coding activated.
INFO  [io.quarkus] (Quarkus Main Thread) Installed features:
  [cdi, funqy-knative-events, smallrye-context-propagation]

--
Tests paused, press [r] to resume
```

> **Note:** Quarkus 2.x provides a continuous testing feature so that you can keep testing your code when you add or update code by pressing r in the terminal.

Now the CloudEvents function is running in your local development environment. So, send a cloud event to the function over the HTTP protocol:

```
curl -v http://localhost:8080 \
  -H "Content-Type:application/json" \
  -H "Ce-Id:1" \
  -H "Ce-Source:cloud-event-example" \
  -H "Ce-Type:myCloudEventGreeting" \
  -H "Ce-Specversion:1.0" \
  -d "{\"name\": \"Daniel\"}"
```

The output should end with:

```
HTTP/1.1 204 No Content
```

Go back to the terminal, and the log should look like this:

```
INFO [org.acm.get.sta.fun.clo.CloudEventGreeting]
  (executor-thread-0) Hello Daniel
```

3. Deploy the serverless event function to Knative

Add a `container-image-docker` extension to the Quarkus Funqy project. The extension enables you to build a container image based on the serverless event function and then push it to an external container registry (e.g., Docker Hub [10], Quay.io [11]):

```
$ ./mvnw quarkus:add-extension -Dextensions="container-image-
    docker"
```

Open the `application.properties` file in the `src/main/resources/` directory. Then add the following variables to configure Knative and Kubernetes resources (make sure to replace `yourAccountName` with your container registry's account name, e.g., your username in Docker Hub):

```
quarkus.container-image.build=true
quarkus.container-image.push=true
quarkus.container-image.builder=docker
quarkus.container-image.image=docker.io/yourAccountName/funqy-
  knative-events-codestart
```

Run the following command to containerize the function and then push it to the Docker Hub container registry automatically:

```
$ ./mvnw clean package
```

The output should end with `BUILD SUCCESS`.

Open the `funqy-service.yaml` file in the `src/main/k8s` directory. Then replace `yourAccountName` with your account information in the Docker Hub registry:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: funqy-knative-events-codestart
spec:
  template:
    metadata:
      name: funqy-knative-events-codestart-v1
      annotations:
        autoscaling.knative.dev/target: "1"
    spec:
      containers:
        - image: docker.io/yourAccountName/funqy-knative-
              events-codestart
```

Assuming the container image pushed successfully, create the Knative service based on the event function using the following `kubectl` command-line tool (be sure to log into the Kubernetes cluster and change the namespace where you want to create the Knative service):

```
$ kubectl create -f src/main/k8s/funqy-service.yaml
```

The output should look like this:

```
service.serving.knative.dev/funqy-knative-events-codestart
  created
```

Create a default broker to subscribe to the event function. Use the kn [12] Knative Serving command-line tool:

```
$ kn broker create default
```

Open the `funqy-trigger.yaml` file in the `src/main/k8s` directory and replace it with:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: my-cloudevent-greeting
spec:
  broker: default
  subscriber:
    ref:
```

```
    apiVersion: serving.knative.dev/v1
    kind: Service
    name: funqy-knative-events-codestart
```

Create a trigger using the `kubectl` command-line tool:

```
$ kubectl create -f src/main/k8s/funqy-trigger.yaml
```

The output should look like this:

```
trigger.eventing.knative.dev/my-cloudevent-greeting created
```

4. Send a cloud event to the serverless event function in Kubernetes

Find out the function's route URL and check that the output looks like this:

```
$ kubectl get rt
NAME URL READY REASON
funqy-knative-events-codestart
  http://funqy-knative-events-codestart-YOUR_HOST_DOMAIN   True
```

Send a cloud event to the function over the HTTP protocol:

```
curl -v http://funqy-knative-events-codestart-YOUR_HOST_DOMAIN \
  -H "Content-Type:application/json" \
  -H "Ce-Id:1" \
  -H "Ce-Source:cloud-event-example" \
  -H "Ce-Type:myCloudEventGreeting" \
  -H "Ce-Specversion:1.0" \
  -d "{\"name\": \"Daniel\"}"
```

The output should end with:

```
HTTP/1.1 204 No Content
```

Once the function pod scales up, take a look at the pod logs. Use the following `kubectl` command to retrieve the pod's name:

```
$ kubectl get pod
```

The output will look like this:

```
NAME                                  READY  STATUS   RESTARTS  AGE
funqy-knative-events-codestart-       2/2    Running  0         11s
  v1-deployment-6569f6dfc-zxsqs
```

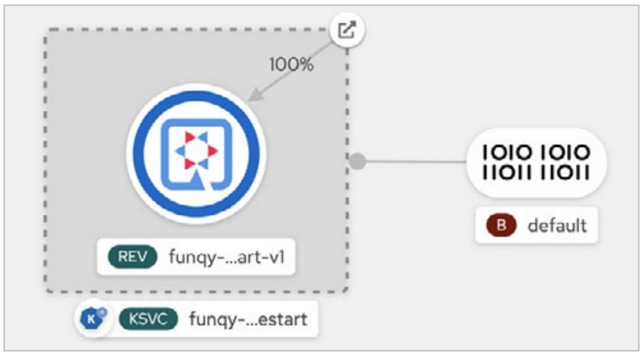
Run the following `kubectl` command to verify that the pod's logs match the local testing's result:

```
$ kubectl logs funqy-knative-events-codestart-v1-deployment-
  6569f6dfc-zxsqs -c user-container | grep CloudEventGreeting
```

The output looks like this:

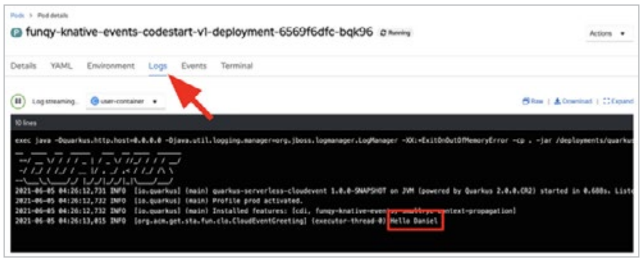
```
INFO  [org.acm.get.sta.fun.clo.CloudEventGreeting]
  (executor-thread-0) Hello Daniel
```

If you deploy the event function to an OpenShift Kubernetes Distribution (OKD) [13] cluster, you will find the deployment status in the topology view:



(Daniel Oh, CC BY-SA 4.0)

You can also find the pod's logs in the **Pod details** tab:



funqy-knative-events-codestart-v1-deployment-6569f6dfc-bqk96

```
exec java -Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager=org.jboss.logmanager.LogManager -XX:+ExitOnOutOfMemoryError -cp . -jar /deployments/quarku
2021-06-05 04:26:12,731 INFO  [io.quarkus] (main) quarkus-serverless-cloudevent 1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.0.0.CR2) started in 0.688s. Liste
2021-06-05 04:26:12,732 INFO  [io.quarkus] (main) Profile prod activated.
2021-06-05 04:26:12,732 INFO  [io.quarkus] (main) Installed features: [cdi, funqy-knative-events, smallrye-context-propagation]
2021-06-05 04:26:13,015 INFO  [org.acm.get.sta.fun.clo.CloudEventGreeting] (executor-thread-0) Hello Daniel
```

(Daniel Oh, CC BY-SA 4.0)

What's next?

Developers can bind a cloud event to Knative using Quarkus functions. Quarkus also scaffolds Kubernetes manifests, such as Knative services and triggers, to process cloud events over a channel or HTTP request.

Learn more serverless and Quarkus topics through OpenShift's interactive self-service learning portal [14].

Links

[1] https://avro.apache.org/
[2] https://developers.google.com/protocol-buffers
[3] https://cloudevents.io/
[4] https://knative.dev/
[5] https://kogito.kie.org/
[6] https://debezium.io/
[7] https://quarkus.io/
[8] https://knative.dev/docs/eventing/
[9] https://opensource.com/article/21/6/quarkus-funqy
[10] https://hub.docker.com/
[11] https://quay.io/
[12] https://knative.dev/docs/client/install-kn/
[13] https://www.okd.io/
[14] https://learn.openshift.com/serverless/