Quarkus - Funqy

Quarkus Funqy is part of Quarkus's serverless strategy and aims to provide a portable Java API to write functions deployable to various FaaS environments like AWS Lambda, Azure Functions, Knative, and Knative Events (Cloud Events). It is also usable as a standalone service.

Because Funqy is an abstraction that spans multiple different cloud/function providers and protocols it has to be a very simple API and thus, might not have all the features you are used to in other remoting abstractions. A nice side effect though is that Funqy is as optimized and as small as possible. This means that because Funqy sacrifices a little bit on flexibility, you'll get a framework that has little to no overhead.

Funqy Basics

The Funqy API is simple. Annotate a method with @Funq. This method may only have one optional input parameter and may or may not return a response.

```
import io.quarkus.funqy.Funq;

public class GreetingFunction {
    @Funq
    public String greet(String name) {
        return "Hello " + name;
    }
}
```

Java classes can also be used as input and output and must follow the Java bean convention and have a default constructor. The Java type that is declared as the parameter or return type is the type that will be expected by the Funqy runtime. Funqy does type introspection at build time to speed up boot time, so any derived types will not be noticed by the Funqy marshalling layer at runtime.

Here's an example of using a POJO as input and output types.

```
public class GreetingFunction {
    public static class Friend {
        String name;

        public String getName() { return name; }
        public void setName(String name) { this.name = name; }
    }

    public static class Greeting {
        String msg;

        public Greeting() {}
        public Greeting(String msg) { this.msg = msg }

        public String getMessage() { return msg; }
        public void setMessage(String msg) { this.msg = msg; }
    }

    @Funq
    public Greeting greet(Friend friend) {
        return new Greeting("Hello " + friend.getName());
    }
}
```

Async Reactive Types

Funqy supports the Smallrye Mutiny Uni reactive type as a return type. The only requirement is that the Uni must fill out the generic type.

```
import io.quarkus.funqy.Funq;
import io.smallrye.mutiny.Uni;

public class GreetingFunction {

    @Funq
    public Uni<Greeting> reactiveGreeting(String name) {
        ...
    }
}
```

Function Names

The function name defaults to the method name and is case sensitive. If you want your function referenced by a different name, parameterize the @Funq annotation as follows:

```
import io.quarkus.funqy.Funq;

public class GreetingFunction {

    @Funq("HelloWorld")
    public String greet(String name) {
        return "Hello " + name;
    }
}
```

Funqy DI

Each Funqy Java class is a Quarkus Arc component and supports dependency injection through CDI or Spring DI. Spring DI requires including the `quarkus-spring-di` dependency in your build.

The default object lifecycle for a Funqy class is `@Dependent`.

```
import io.quarkus.funqy.Funq;

import javax.inject.Inject;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class GreetingFunction {

    @Inject
    GreetingService service;

    @Funq
    public Greeting greet(Friend friend) {
        Greeting greeting = new Greeting();
        greeting.setMessage(service.greet(friend.getName()));
        return greeting;
    }
}
```

Context injection

The Funqy API will usually not allow you to inject or use abstractions that are specific to a protocol (i.e. HTTP) or function API (i.e. AWS Lambda). There are exceptions to the rule though and you may be able to inject contextual information that is specific to the environment you are deploying in.

> ℹ️ We do not recommend injecting contextual information specific to a runtime. Keep your functions portable.

Contextual information is injected via the @Context annotation which can be used on a function parameter or a class field. A good example is the CloudEvent interface that comes with our Funqy Knative Cloud Events integration:

```java
import io.quarkus.funqy.Funq;
import io.quarkus.funqy.Context;

public class GreetingFunction {

    @Funq
    public Greeting greet(Friend friend, @Context CloudEvent eventInfo) {
        System.out.println("Received greeting request from: " eventInfo.getSource());

        Greeting greeting = new Greeting();
        greeting.setMessage("Hello " + friend.getName()));
        return greeting;
    }
}
```

Should I Use Funqy?

REST over HTTP has become a very common way to write services over the past decade. While Funqy has an HTTP binding it is not a replacement for REST. Because Funqy has to work across a variety of protocols and function cloud platforms, it is very minimalistic and constrained. For example, if you use Funqy you lose the ability to link (think URIs) to the data your functions spit out. You also lose the ability to leverage cool HTTP features like cache-control and conditional GETs. Many developers will be ok with that as many won't be using these REST/HTTP features or styles. You'll have to make the decision on what camp you are in. Quarkus does support REST integration (through JAX-RS, Spring MVC, Vert.x Web, and Servlet) with various cloud/function providers, but there are some disadvantages of using that approach as well. For example, if you want to do HTTP with AWS Lambda, this requires you to use the AWS API Gateway which may slow down deployment and cold start time or even cost you more.

The purpose of Funqy is to allow you to write cross-provider functions so that you can move off of your current function provider if, for instance, they start charging you a lot more for their service. Another reason you might not want to use Funqy is if you need access specific APIs of the target function environment. For example, developers often want access to the AWS Context on Lambda. In this case, we tell them they may be better off using the Quarkus Amazon Lambda integration instead.