

Quarkus - Funqy HTTP Binding (Standalone)

The guide walks through quickstart code to show you how you can deploy Funqy as a standalone service and invoke on Funqy functions using HTTP.



The Funqy HTTP binding is not a replacement for REST over HTTP. Because Funqy needs to be portable across a lot of different protocols and function providers its HTTP binding is very minimalistic and you will lose REST features like linking and the ability to leverage HTTP features like cache-control and conditional GETs. You may want to consider using Quarkus's JAX-RS, Spring MVC, or Vert.x Web Reactive Routes support instead, although Funqy will have less overhead than these alternatives (except Vert.x which is still super fast).

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- Read about [Funqy Basics](#). This is a short read!
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3

The Quickstart

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `funqy-http-quickstart` directory.

The Code

If you look at the Java code, you'll see that there is no HTTP specific API. Its just simple Java methods annotated with `@Funq`. Simple, easy, straightforward.

Maven Dependencies

To write Funqy HTTP functions, simply include the `quarkus-funqy-http` dependency into your Quarkus `pom.xml` file:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-funqy-http</artifactId>
</dependency>
```

Build Project

```
mvn clean quarkus:dev
```

This starts your functions in Quarkus dev mode.

Execute Funqy HTTP functions

The URL path to execute a function is the function name. For example if your function name is `foo` then the URL path to execute the function would be `/foo`.

The HTTP POST or GET methods can be used to invoke on a function. The return value of the function is marshalled to JSON using the Jackson JSON library. Jackson annotations can be used. If your function has an input parameter, a POST invocation must use JSON as the input type. Jackson is also used here for unmarshalling.

You can invoke the `hello` function defined in [PrimitiveFunctions.java](#) by pointing your browser to <http://localhost:8080/hello>

Invoking the other functions in the quickstart requires an HTTP POST. To execute the `greet` function defined in [GreetingFunction.java](#) invoke this curl script.

```
curl "http://localhost:8080/greet" \
-X POST \
-H "Content-Type: application/json" \
-d '{"name": "Bill"}'
```

Primitive types can also be passed as input using the standard JSON mapping for them. To execute the `toLowerCase` function defined in [PrimitiveFunctions.java](#) invoke this curl script:

```
curl "http://localhost:8080/toLowerCase" \
-X POST \
-H "Content-Type: application/json" \
-d '"HELLO WORLD"'
```

To execute the `double` function defined in [PrimitiveFunctions.java](#) invoke this curl script:

```
curl "http://localhost:8080/double" \  
-X POST \  
-H "Content-Type: application/json" \  
-d '2'
```

GET Query Parameter Mapping

For GET requests, the Funqy HTTP Binding also has a query parameter mapping for function input parameters. Only bean style classes and `java.util.Map` can be used for your input parameter. For bean style classes, query parameter names are mapped to properties on the bean class. Here's an example of a simple `Map`:

```
@Funq  
public String hello(Map<String, Integer> map) {  
    ...  
}
```

Key values must be a primitive type (except char) or `String`. Values can be primitives (except char), `String`, `OffsetDateTime` or a complex bean style class. For the above example, here's the corresponding curl request:

```
curl "http://localhost:8080/a=1&b=2"
```

The `map` input parameter of the `hello` function would have the key value pairs: `a→1`, `b→2`.

Bean style classes can also be use as the input parameter type. Here's an example:

```
public class Person {  
    String first;  
    String last;  
  
    public String getFirst() { return first; }  
    public void setFirst(String first) { this.first = first; }  
    public String getLast() { return last; }  
    public void setLast(String last) { this.last = last; }  
}  
  
public class MyFunctions {  
    @Funq  
    public String greet(Person p) {  
        return "Hello " + p.getFirst() + " " + p.getLast();  
    }  
}
```

Property values can be any primitive type except `char`. It can also be `String`, and `OffsetDateTime`. `OffsetDateTime` query param values must be in ISO-8601 format.

You can invoke on this using an HTTP GET and query parameters:

```
curl "http://localhost:8080/greet?first=Bill&last=Burke"
```

In the above request, the query parameter names are mapped to corresponding properties in the input class.

The input class can also have nested bean classes. Expanding on the previous example:

```
public class Family {
    private Person dad;
    private Person mom;

    public Person getDad() { return dad; }
    public void setDad(Person dad) { this.dad = dad; }
    public Person getMom() { return mom; }
    public void setMom(Person mom) { this.mom = mom; }
}

public class MyFunctions {
    @Funq
    public String greet(Family family) {
        ...
    }
}
```

In this case, query parameters for nested values use the `.` notation. For example:

```
curl
"http://localhost:8080/greet?dad.first=John&dad.last=Smith&mom.firs
t=Martha&mom.last=Smith"
```

`java.util.List` and `Set` are also supported as property values. For example:

```

public class Family {
    ...

    List<String> pets;
}

public class MyFunctions {
    @Func
    public String greet(Family family) {
        ...
    }
}

```

To invoke a GET request, just list the `pets` query parameter multiple times.

```
curl "http://localhost:8080/greet?pets=itchy&pets=scratchy"
```

For more complex types, `List` and `Set` members must have an identifier in the query parameter. For example:

```

public class Family {
    ...

    List<Person> kids;
}

public class MyFunctions {
    @Func
    public String greet(Family family) {
        ...
    }
}

```

Each `kids` query parameter must identify the kid they are referencing so that the runtime can figure out which property values go to which members in the list. Here's the curl request:

```

curl
"http://localhost:8080/greet?kids.1.first=Buffy&kids.2.first=Charli
e"

```

The above URL uses the value `1` and `2` to identify the target member of the list, but any unique string can be used.

A property can also be a `java.util.Map`. The key of the map can be any primitive type and `String`. For example:

```

public class Family {
    ...

    Map<String, String> address;
}

public class MyFunctions {
    @Funq
    public String greet(Family family) {
        ...
    }
}

```

The corresponding call would look like this:

```

curl
"http://localhost:8080/greet?address.state=MA&address.city=Boston"

```

If your **Map** value is a complex type, then just continue the notation by adding the property to set at the end.

```

public class Family {
    ...

    Map<String, Address> addresses;
}

public class MyFunctions {
    @Funq
    public String greet(Family family) {
        ...
    }
}

```

```

curl
"http://localhost:8080/greet?addresses.home.state=MA&addresses.home.city=Boston"

```