Quarkus - Using the REST Client

This guide explains how to use the MicroProfile REST Client in order to interact with REST APIs with very little effort.

> 💡   there is another guide if you need to write server JSON REST APIs.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with JAVA_HOME configured appropriately
- Apache Maven 3.6.3

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: git clone https://github.com/quarkusio/quarkus-quickstarts.git, or download an archive.

The solution is located in the rest-client-quickstart directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=rest-client-quickstart \
    -DclassName="org.acme.rest.client.CountriesResource" \
    -Dpath="/country" \
    -Dextensions="rest-client, resteasy-jsonb"
cd rest-client-quickstart
```

This command generates the Maven project with a REST endpoint and imports the rest-client and resteasy-jsonb extensions.

If your application does not expose any JAX-RS endpoints (eg. a command mode application), use the `rest-client-jsonb` or the `rest-client-jackson` extension instead.

If you already have your Quarkus project configured, you can add the `rest-client` and the `rest-client-jsonb` extensions to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="rest-client,resteasy-jsonb"
```

This will add the following to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-rest-client</artifactId>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-jsonb</artifactId>
</dependency>
```

Setting up the model

In this guide we will be demonstrating how to consume part of the REST API supplied by the restcountries.eu service. Our first order of business is to setup the model we will be using, in the form of a `Country` POJO.

Create a `src/main/java/org/acme/rest/client/Country.java` file and set the following content:

```
package org.acme.rest.client;

import java.util.List;

public class Country {

    public String name;
    public String alpha2Code;
    public String capital;
    public List<Currency> currencies;

    public static class Currency {
        public String code;
        public String name;
        public String symbol;
    }
}
```

The model above is only a subset of the fields provided by the service, but it suffices for the purposes of this guide.

Create the interface

Using the MicroProfile REST Client is as simple as creating an interface using the proper JAX-RS and MicroProfile annotations. In our case the interface should be created at src/main/java/org/acme/rest/client/CountriesService.java and have the following content:

```
package org.acme.rest.client;

import
org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import java.util.Set;

@Path("/v2")
@RegisterRestClient
public interface CountriesService {

    @GET
    @Path("/name/{name}")
    @Produces("application/json")
    Set<Country> getByName(@PathParam String name);
}
```

The `getByName` method gives our code the ability to query a country by name from the REST Countries API. The client will handle all the networking and marshalling leaving our code clean of such technical details.

The purpose of the annotations in the code above is the following:

- `@RegisterRestClient` allows Quarkus to know that this interface is meant to be available for CDI injection as a REST Client

- `@Path`, `@GET` and `@PathParam` are the standard JAX-RS annotations used to define how to access the service

- `@Produces` defines the expected content-type

> ℹ️ While `@Consumes` and `@Produces` are optional as auto-negotiation is supported, it is heavily recommended to annotate your endpoints with them to define precisely the expected content-types.
>
> It will allow to narrow down the number of JAX-RS providers (which can be seen as converters) included in the native executable.

Create the configuration

In order to determine the base URL to which REST calls will be made, the REST Client uses configuration from `application.properties`. The name of the property needs to follow a certain convention which is best displayed in the following code:

```
# Your configuration properties
org.acme.rest.client.CountriesService/mp-
rest/url=https://restcountries.eu/rest # ①
org.acme.rest.client.CountriesService/mp-
rest/scope=javax.inject.Singleton # ②
```

① Having this configuration means that all requests performed using `org.acme.rest.client.CountriesService` will use `https://restcountries.eu/rest` as the base URL. Using the configuration above, calling the `getByName` method of `CountriesService` with a value of `France` would result in an HTTP GET request being made to `https://restcountries.eu/rest/v2/name/France`.

② Having this configuration means that the default scope of `org.acme.rest.client.CountriesService` will be `@Singleton`. Supported scope values are `@Singleton`, `@Dependent`, `@ApplicationScoped` and `@RequestScoped`. The default scope is `@Dependent`. The default scope can also be defined on the interface.

Note that `org.acme.rest.client.CountriesService` *must* match the fully qualified name of the `CountriesService` interface we created in the previous section.

To facilitate the configuration, you can use the `@RegisterRestClient configKey` property that allows to use another configuration root than the fully qualified name of your interface.

```
@RegisterRestClient(configKey="country-api")
public interface CountriesService {
    [...]
}
```

```
# Your configuration properties
country-api/mp-rest/url=https://restcountries.eu/rest #
country-api/mp-rest/scope=javax.inject.Singleton # /
```

Update the JAX-RS resource

Open the `src/main/java/org/acme/rest/client/CountriesResource.java` file and update it with the following content:

```java
import org.eclipse.microprofile.rest.client.inject.RestClient;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.Set;

@Path("/country")
public class CountriesResource {

    @Inject
    @RestClient
    CountriesService countriesService;


    @GET
    @Path("/name/{name}")
    @Produces(MediaType.APPLICATION_JSON)
    public Set<Country> name(@PathParam String name) {
        return countriesService.getByName(name);
    }
}
```

Note that in addition to the standard CDI `@Inject` annotation, we also need to use the MicroProfile `@RestClient` annotation to inject `CountriesService`.

Update the test

We also need to update the functional test to reflect the changes made to the endpoint. Edit the `src/test/java/org/acme/rest/client/CountriesResourceTest.java` file and change the content of the `testCountryNameEndpoint` method to:

```java
package org.acme.rest.client;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class CountriesResourceTest {

    @Test
    public void testCountryNameEndpoint() {
        given()
          .when().get("/country/name/greece")
          .then()
             .statusCode(200)
             .body("$.size()", is(1),
                     "[0].alpha2Code", is("GR"),
                     "[0].capital", is("Athens"),
                     "[0].currencies.size()", is(1),
                     "[0].currencies[0].name", is("Euro")
             );
    }

}
```

The code above uses REST Assured's json-path capabilities.

Async Support

The rest client supports asynchronous rest calls. Async support comes in 2 flavors: you can return a `CompletionStage` or a `Uni` (requires the `quarkus-resteasy-mutiny` extension). Let's see it in action by adding a `getByNameAsync` method in our `CountriesService` REST interface. The code should look like:

```
package org.acme.rest.client;

import
org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import java.util.concurrent.CompletionStage;
import java.util.Set;

@Path("/v2")
@RegisterRestClient
public interface CountriesService {

    @GET
    @Path("/name/{name}")
    @Produces("application/json")
    Set<Country> getByName(@PathParam String name);

    @GET
    @Path("/name/{name}")
    @Produces("application/json")
    CompletionStage<Set<Country>> getByNameAsync(@PathParam String
name);
}
```

Open the `src/main/java/org/acme/rest/client/CountriesResource.java` file and update it with the following content:

```java
package org.acme.rest.client;

import org.eclipse.microprofile.rest.client.inject.RestClient;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.concurrent.CompletionStage;
import java.util.Set;

@Path("/country")
public class CountriesResource {

    @Inject
    @RestClient
    CountriesService countriesService;


    @GET
    @Path("/name/{name}")
    @Produces(MediaType.APPLICATION_JSON)
    public Set<Country> name(@PathParam String name) {
        return countriesService.getByName(name);
    }

    @GET
    @Path("/name-async/{name}")
    @Produces(MediaType.APPLICATION_JSON)
    public CompletionStage<Set<Country>> nameAsync(@PathParam
String name) {
        return countriesService.getByNameAsync(name);
    }
}
```

To test asynchronous methods, add the test method below in `CountriesResourceTest`:

```
@Test
public void testCountryNameAsyncEndpoint() {
    given()
    .when().get("/country/name-async/greece")
    .then()
        .statusCode(200)
        .body("$.size()", is(1),
            "[0].alpha2Code", is("GR"),
            "[0].capital", is("Athens"),
            "[0].currencies.size()", is(1),
            "[0].currencies[0].name", is("Euro")
        );
    }
```

The Uni version is very similar:

```
package org.acme.rest.client;

import
org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import java.util.concurrent.CompletionStage;
import java.util.Set;

@Path("/v2")
@RegisterRestClient
public interface CountriesService {

    // ...

    @GET
    @Path("/name/{name}")
    @Produces("application/json")
    Uni<Set<Country>> getByNameAsUni(@PathParam String name);
}
```

The CountriesResource becomes:

```
package org.acme.rest.client;

import org.eclipse.microprofile.rest.client.inject.RestClient;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.concurrent.CompletionStage;
import java.util.Set;

@Path("/country")
public class CountriesResource {

    @Inject
    @RestClient
    CountriesService countriesService;


    // ...

    @GET
    @Path("/name-uni/{name}")
    @Produces(MediaType.APPLICATION_JSON)
    public Uni<Set<Country>> nameAsync(@PathParam String name) {
        return countriesService.getByNameAsUni(name);
    }
}
```

> *Mutiny*
>
> The previous snippet uses Mutiny reactive types, if you're not familiar with them, read the Getting Started with Reactive guide first.

When returning a `Uni`, every *subscription* invokes the remote service. It means you can re-send the request by re-subscribing on the `Uni`, or use a `retry` as follow:

```
@Inject @RestClient CountriesResource service;

// ...

service.nameAsync("Greece")
    .onFailure().retry().atMost(10);
```

If you use a `CompletionStage`, you would need to call the service's method to retry. This difference comes from the laziness aspect of Mutiny and its subscription protocol. More details about this can be

found in [the Mutiny documentation](#).

Package and run the application

Run the application with: `./mvnw compile quarkus:dev`. Open your browser to http://localhost:8080/country/name/greece.

You should see a JSON object containing some basic information about Greece.

As usual, the application can be packaged using `./mvnw clean package` and executed using the `-runner.jar` file. You can also generate the native executable with `./mvnw clean package -Pnative`.

Using a Mock HTTP Server for tests

Setting up a mock HTTP server, against which tests are run, is a common testing pattern. Examples of such servers are [Wiremock](#) and [Hoverfly](#). In this section we'll demonstrate how Wiremock can be leveraged for testing the `CountriesService` which was developed above.

First of all, Wiremock needs to be added as a test dependency. For a Maven project that would happen like so:

```xml
<dependency>
    <groupId>com.github.tomakehurst</groupId>
    <artifactId>wiremock-jre8</artifactId>
    <scope>test</scope>
    <version>${wiremock.version}</version> ①
</dependency>
```

① Use a proper Wiremock version. All available versions can be found [here](#).

In Quarkus tests when some service needs to be started before the Quarkus tests are ran, we utilize the `@io.quarkus.test.common.QuarkusTestResource` annotation to specify a `io.quarkus.test.common.QuarkusTestResourceLifecycleManager` which can start the service and supply configuration values that Quarkus will use.

> **ⓘ** For more details about `@QuarkusTestResource` refer to [this part of the documentation](#).

Let's create an implementation of `QuarkusTestResourceLifecycleManager` called `WiremockCountries` like so:

```java
package org.acme.rest.client;

import java.util.Collections;
import java.util.Map;

import com.github.tomakehurst.wiremock.WireMockServer;
import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;

import static com.github.tomakehurst.wiremock.client.WireMock.*;  ①

public class WiremockCountries implements
QuarkusTestResourceLifecycleManager {  ②

    private WireMockServer wireMockServer;

    @Override
    public Map<String, String> start() {
        wireMockServer = new WireMockServer();
        wireMockServer.start();  ③

        stubFor(get(urlEqualTo("/v2/name/GR"))    ④
                .willReturn(aResponse()
                        .withHeader("Content-Type",
"application/json")
                        .withBody(
                        "[{" +
                        "\"name\": \"Ελλάδα\"," +
                        "\"capital\": \"Αθήνα\"" +
                        "}]"
                        )));


stubFor(get(urlMatching(".*")).atPriority(10).willReturn(aResponse(
).proxiedFrom("https://restcountries.eu/rest")));    ⑤

        return
Collections.singletonMap("org.acme.getting.started.country.Countrie
sService/mp-rest/url", wireMockServer.baseUrl());  ⑥
    }

    @Override
    public void stop() {
        if (null != wireMockServer) {
            wireMockServer.stop();  ⑦
        }
    }
}
```

① Statically importing the methods in the Wiremock package makes it easier to read the test.

② The `start` method is invoked by Quarkus before any test is run and returns a `Map` of configuration properties that apply during the test execution.

③ Launch Wiremock.

④ Configure Wiremock to stub the calls to `/v2/name/GR` by returning a specific canned response.

⑤ All HTTP calls that have not been stubbed are handled by calling the real service. This is done for demonstration purposes, as it is not something that would usually happen in a real test.

⑥ As the `start` method returns configuration that applies for tests, we set the rest-client property that controls the base URL which is used by the implementation of `CountriesService` to the base URL where Wiremock is listening for incoming requests.

⑦ When all tests have finished, shutdown Wiremock.

The `CountriesResourceTest` test class needs to be annotated like so:

```
@QuarkusTest
@QuarkusTestResource(WiremockCountries.class)
public class CountriesResourceTest {

}
```

⚠️ `@QuarkusTestResource` applies to all tests, not just `CountriesResourceTest`.

Further reading

• MicroProfile Rest Client specification