Quarkus - Using the Redis Client

This guide demonstrates how your Quarkus application can connect to a Redis server using the Redis Client extension.

This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.

For a full list of possible extension statuses, check our FAQ entry.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+
- A running Redis server, or Docker Compose to start one
- GraalVM installed if you want to run in native mode.

Architecture

In this guide, we are going to expose a simple Rest API to increment numbers by using the `INCRBY` command. Along the way, we'll see how to use other Redis commands like `GET`, `SET`, `DEL` and `KEYS`.

We'll be using the Quarkus Redis Client extension to connect to our Redis Server. The extension is implemented on top of the Vert.x Redis Client, providing an asynchronous and non-blocking way to connect to Redis.

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 `redis-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:1.7.2.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=redis-quickstart \
    -Dextensions="redis-client, resteasy-jsonb, resteasy-mutiny"
cd redis-quickstart
```

This command generates a Maven project, importing the Redis extension.

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

```
./mvnw quarkus:add-extension -Dextensions="redis-client"
```

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

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-redis-client</artifactId>
</dependency>
```

Starting the Redis server

Then, we need to start a Redis instance (if you do not have one already) using the following command:

```
docker run --ulimit memlock=-1:-1 -it --rm=true --memory
-swappiness=0 --name redis_quarkus_test -p 6379:6379 redis:5.0.6
```

Configuring Redis properties

Once we have the Redis server running, we need to configure the Redis connection properties. This is done in the `application.properties` configuration file. Edit it to the following content:

```
quarkus.redis.hosts=localhost:6379  ①
```

1. Configure Redis hosts to connect to. Here we connect to the Redis server we started in the previous section

Creating the Increment POJO

We are going to model our increments using the `Increment` POJO. Create the `src/main/java/org/acme/redis/Increment.java` file, with the following content:

```java
package org.acme.redis;

public class Increment {
    public String key; ①
    public int value; ②

    public Increment(String key, int value) {
        this.key = key;
        this.value = value;
    }

    public Increment() {
    }
}
```

1. The key that will be used as the Redis key

2. The value held by the Redis key

Creating the Increment Service

We are going to create an `IncrementService` class which will play the role of a Redis client. With this class, we'll be able to perform the `SET`, `GET`, `DELET`, `KEYS` and `INCRBY` Redis commands.

Create the `src/main/java/org/acme/redis/IncrementService.java` file, with the following content:

```java
package org.acme.redis;

import io.quarkus.redis.client.RedisClient;
import io.quarkus.redis.client.reactive.ReactiveRedisClient;
import io.smallrye.mutiny.Uni;

import io.vertx.mutiny.redis.client.Response;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import javax.inject.Inject;
import javax.inject.Singleton;
```

```
@Singleton
class IncrementService {

    @Inject
    RedisClient redisClient; ①

    @Inject
    ReactiveRedisClient reactiveRedisClient; ②

    Uni<Void> del(String key) {
        return reactiveRedisClient.del(Arrays.asList(key))
                .map(response -> null);
    }

    String get(String key) {
        return redisClient.get(key).toString();
    }

    void set(String key, Integer value) {
        redisClient.set(Arrays.asList(key, value.toString()));
    }

    void increment(String key, Integer incrementBy) {
        redisClient.incrby(key, incrementBy.toString());
    }

    Uni<List<String>> keys() {
        return reactiveRedisClient
                .keys("*")
                .map(response -> {
                    List<String> result = new ArrayList<>();
                    for (Response r : response) {
                        result.add(r.toString());
                    }
                    return result;
                });
    }
}
```

1. Inject the Redis synchronous client

2. Inject the Reactive Redis client

Creating the Increment Resource

Create the `src/main/java/org/acme/redis/IncrementResource.java` file, with the following content:

```java
package org.acme.redis;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.PUT;
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
import javax.ws.rs.POST;
import javax.ws.rs.DELETE;
import javax.ws.rs.core.MediaType;
import java.util.List;

import io.smallrye.mutiny.Uni;

@Path("/increments")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class IncrementResource {

    @Inject
    IncrementService service;

    @GET
    public Uni<List<String>> keys() {
        return service.keys();
    }

    @POST
    public Increment create(Increment increment) {
        service.set(increment.key, increment.value);
        return increment;
    }

    @GET
    @Path("/{key}")
    public Increment get(@PathParam("key") String key) {
        return new Increment(key,
Integer.valueOf(service.get(key)));
    }

    @PUT
    @Path("/{key}")
    public void increment(@PathParam("key") String key, Integer
value) {
        service.increment(key, value);
    }

    @DELETE
```

```
        @Path("/{key}")
        public Uni<Void> delete(@PathParam("key") String key) {
            return service.del(key);
        }
    }
```

Modifying the test class

Edit the `src/test/java/org/acme/redis/IncrementResourceTest.java` file to the following content:

```
package org.acme.redis;

import static org.hamcrest.Matchers.is;

import org.junit.jupiter.api.Test;

import io.quarkus.test.junit.QuarkusTest;

import static io.restassured.RestAssured.given;

import io.restassured.http.ContentType;

@QuarkusTest
public class IncrementResourceTest {

    @Test
    public void testRedisOperations() {
        // verify that we have nothing
        given()
                .accept(ContentType.JSON)
                .when()
                .get("/increments")
                .then()
                .statusCode(200)
                .body("size()", is(0));

        // create a first increment key with an initial value of 0
        given()
                .contentType(ContentType.JSON)
                .accept(ContentType.JSON)
                .body("{\"key\":\"first-key\",\"value\":0}")
                .when()
                .post("/increments")
                .then()
                .statusCode(200)
                .body("key", is("first-key"))
                .body("value", is(0));
```

```java
        // create a second increment key with an initial value of
10
        given()
                .contentType(ContentType.JSON)
                .accept(ContentType.JSON)
                .body("{\"key\":\"second-key\",\"value\":10}")
                .when()
                .post("/increments")
                .then()
                .statusCode(200)
                .body("key", is("second-key"))
                .body("value", is(10));

        // increment first key by 1
        given()
                .contentType(ContentType.JSON)
                .body("1")
                .when()
                .put("/increments/first-key")
                .then()
                .statusCode(204);

        // verify that key has been incremented
        given()
                .accept(ContentType.JSON)
                .when()
                .get("/increments/first-key")
                .then()
                .statusCode(200)
                .body("key", is("first-key"))
                .body("value", is(1));

        // increment second key by 1000
        given()
                .contentType(ContentType.JSON)
                .body("1000")
                .when()
                .put("/increments/second-key")
                .then()
                .statusCode(204);

        // verify that key has been incremented
        given()
                .accept(ContentType.JSON)
                .when()
                .get("/increments/second-key")
                .then()
                .statusCode(200)
```

```java
                    .body("key", is("second-key"))
                    .body("value", is(1010));

        // verify that we have two keys in registered
        given()
                .accept(ContentType.JSON)
                .when()
                .get("/increments")
                .then()
                .statusCode(200)
                .body("size()", is(2));

        // delete first key
        given()
                .accept(ContentType.JSON)
                .when()
                .delete("/increments/first-key")
                .then()
                .statusCode(204);

        // verify that we have one key left after deletion
        given()
                .accept(ContentType.JSON)
                .when()
                .get("/increments")
                .then()
                .statusCode(200)
                .body("size()", is(1));

        // delete second key
        given()
                .accept(ContentType.JSON)
                .when()
                .delete("/increments/second-key")
                .then()
                .statusCode(204);

        // verify that there is no key left
        given()
                .accept(ContentType.JSON)
                .when()
                .get("/increments")
                .then()
                .statusCode(200)
                .body("size()", is(0));
    }
}
```

Get it running

If you followed the instructions, you should have the Redis server running. Then, you just need to run the application using:

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

Open another terminal and run the `curl http://localhost:8080/increments` command.

Interacting with the application

As we have seen above, the API exposes five Rest endpoints. In this section we are going to see how to initialise an increment, see the list of current increments, incrementing a value given its key, retrieving the current value of an increment, and finally deleting a key.

Creating a new increment

```
curl -X POST -H "Content-Type: application/json" -d
'{"key":"first","value":10}' http://localhost:8080/increments  ①
```

1. We create the first increment, with the key `first` and an initial value of `10`.

Running the above command should return the result below:

```
{
  "key": "first",
  "value": 10
}
```

See current increments keys

To see the list of current increments keys, run the following command:

```
curl http://localhost:8080/increments
```

The above command should return `["first"]` indicating that we have only one increment thus far.

Retrieve a new increment

To retrieve an increment using its key, we will have to run the below command:

```
curl http://localhost:8080/increments/first ①
```

1. Running this command, should return the following result:

```
{
  "key": "first",
  "value": 10
}
```

Increment a value given its key

To increment a value, run the following command:

```
curl -X PUT -H "Content-Type: application/json" -d '27'
http://localhost:8080/increments/first ①
```

1. Increment the `first` value by 27.

Now, running the command `curl http://localhost:8080/increments/first` should return the following result:

```
{
  "key": "first",
  "value": 37 ①
}
```

1. We see that the value of the `first` key is now 37 which is exactly the result of `10 + 27`, quick maths.

Deleting a key

Use the command below, to delete an increment given its key.

```
curl -X DELETE  http://localhost:8080/increments/first ①
```

1. Delete the `first` increment.

Now, running the command `curl http://localhost:8080/increments` should return an empty list `[]`

Packaging and running in JVM mode

You can run the application as a conventional jar file.

First, we will need to package it:

```
./mvnw package
```

> **ℹ** This command will start a Redis instance to execute the tests. Thus your Redis containers need to be stopped.

Then run it:

```
java -jar ./target/redis-quickstart-1.0-SNAPSHOT-runner.jar
```

Running Native

You can also create a native executable from this application without making any source code changes. A native executable removes the dependency on the JVM: everything needed to run the application on the target platform is included in the executable, allowing the application to run with minimal resource overhead.

Compiling a native executable takes a bit longer, as GraalVM performs additional steps to remove unnecessary codepaths. Use the `native` profile to compile a native executable:

```
./mvnw package -Pnative
```

Once the build is finished, you can run the executable with:

```
./target/redis-quickstart-1.0-SNAPSHOT-runner
```

Connection Health Check

If you are using the `quarkus-smallrye-health` extension, `quarkus-vertx-redis` will automatically add a readiness health check to validate the connection to the Redis server.

So when you access the `/health/ready` endpoint of your application you will have information about the connection validation status.

This behavior can be disabled by setting the `quarkus.redis.health.enabled` property to `false` in your `application.properties`.

Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.redis.health.enabled` Whether or not an health check is published in case the smallrye-health extension is present.	boolean	`true`
`quarkus.redis.password` The redis password	string	
`quarkus.redis.hosts` The redis hosts	list of host:port	`localhost:6379`
`quarkus.redis.database` The redis database	int	`0`
`quarkus.redis.timeout` The maximum delay to wait before a blocking command to redis server times out	Duration ❓	`10s`
`quarkus.redis.ssl` Enables or disables the SSL on connect.	boolean	`false`
`quarkus.redis.client-type` The redis client type	`standalone`, `sentinel`, `cluster`	`standalone`
`quarkus.redis.max-pool-size` The maximum size of the connection pool. When working with cluster or sentinel. This value should be at least the total number of cluster member (or number of sentinels + 1)	int	`6`
`quarkus.redis.max-pool-waiting` The maximum waiting requests for a connection from the pool.	int	`24`

`quarkus.redis.pool-cleaner-interval` The duration indicating how often should the connection pool cleaner executes.	Duration ❓	
`quarkus.redis.pool-recycle-timeout` The timeout for a connection recycling.	Duration ❓	15

ℹ️ *About the Duration format*

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.