Quarkus - Reactive SQL Clients

The Reactive SQL Clients have a straightforward API focusing on scalability and low-overhead. Currently, the following database servers are supported:

- PostgreSQL
- MySQL

In this guide, you will learn how to implement a simple CRUD application exposing data stored in **PostgreSQL** over a RESTful API.

> ℹ️ Extension and connection pool class names for each client can be found at the bottom of this document.

> ❗ If you are not familiar with the Quarkus Vert.x extension, consider reading the Using Eclipse Vert.x guide first.

The application shall manage fruit entities:

```java
public class Fruit {

    public Long id;

    public String name;

    public Fruit() {
    }

    public Fruit(String name) {
        this.name = name;
    }

    public Fruit(Long id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Do you need a ready-to-use PostgreSQL server to try out the examples?

```
docker run --ulimit memlock=-1:-1 -it --rm=true
--memory-swappiness=0 --name quarkus_test -e
POSTGRES_USER=quarkus_test -e
POSTGRES_PASSWORD=quarkus_test -e
POSTGRES_DB=quarkus_test -p 5432:5432 postgres:10.5
```

Installing

Reactive PostgreSQL Client extension

First, make sure your project has the `quarkus-reactive-pg-client` extension enabled. If you are creating a new project, set the `extensions` parameter as follows:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=reactive-pg-client-quickstart \
    -Dextensions="reactive-pg-client"
cd reactive-pg-client-quickstart
```

If you have an already created project, the `reactive-pg-client` extension can be added to an existing Quarkus project with the `add-extension` command:

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

Otherwise, you can manually add this to the dependencies section of your `pom.xml` file:

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

In this guide, we will use the Axle API of the Reactive PostgreSQL Client. Read the Using Eclipse Vert.x guide to understand the differences between the callback, Mutiny, RxJava and Axle based APIs. The RxJava and Axle API are deprecated, and are planned for removal. It is recommended to switch to Mutiny.

Mutiny

The recommended API uses Mutiny reactive types, if you're not familiar with them, read the Getting Started with Reactive guide first.

JSON Binding

We will expose `Fruit` instances over HTTP in the JSON format. Consequently, you also need to add the `quarkus-resteasy-jsonb` extension:

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

If you prefer not to use the command line, manually add this to the dependencies section of your `pom.xml` file:

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

Of course, this is only a requirement for this guide, not any application using the Reactive PostgreSQL Client.

Configuring

The Reactive PostgreSQL Client can be configured with standard Quarkus datasource properties:

src/main/resources/application.properties

```
quarkus.datasource.url=vertx-
reactive:postgresql://localhost:5432/quarkus_test
quarkus.datasource.username=quarkus_test
quarkus.datasource.password=quarkus_test
```

> The eagle eyes among you might have noticed the `url` prefix is different from usual JDBC urls. It is required to use the `vertx-reactive:` prefix otherwise the Reactive PostgreSQL Client extension will not use the `quarkus.datasource.url` config property.

With that you may create your `FruitResource` skeleton and `@Inject` a `io.vertx.mutiny.pgclient.PgPool` instance:

```java
@Path("fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitResource {

    @Inject
    io.vertx.mutiny.pgclient.PgPool client;
}
```

Database schema and seed data

Before we implement the REST endpoint and data management code, we need to setup the database schema. It would also be convenient to have some data inserted upfront.

For production we would recommend to use something like the Flyway database migration tool. But for development we can simply drop and create the tables on startup, and then insert a few fruits.

src/main/java/org/acme/vertx/FruitResource.java

```java
    @Inject
    @ConfigProperty(name = "myapp.schema.create", defaultValue =
"true") ①
    boolean schemaCreate;

    @PostConstruct
    void config() {
        if (schemaCreate) {
            initdb();
        }
    }

    private void initdb() {
        // TODO
    }
}
```

> 💡 You may override the default value of the `myapp.schema.create` property in the `application.properties` file.

Almost ready! To initialize the DB in development mode, we will use the client simple `query` method. It returns a `Uni` and thus can be composed to execute queries sequentially:

```
 client.query("DROP TABLE IF EXISTS fruits")
    .flatMap(r -> client.query("CREATE TABLE fruits (id SERIAL
PRIMARY KEY, name TEXT NOT NULL)"))
    .flatMap(r -> client.query("INSERT INTO fruits (name) VALUES
('Orange')"))
    .flatMap(r -> client.query("INSERT INTO fruits (name) VALUES
('Pear')"))
    .flatMap(r -> client.query("INSERT INTO fruits (name) VALUES
('Apple')"))
    .await().indefinitely();
```

> **ⓘ** Wondering why we need block until the latest query is completed? This code is part of a `@PostConstruct` method and Quarkus invokes it synchronously. As a consequence, returning prematurely could lead to serving requests while the database is not ready yet.

That's it! So far we have seen how to configure a pooled client and execute simple queries. We are now ready to develop the data management code and implement our RESTful endpoint.

Using

Query results traversal

In development mode, the database is set up with a few rows in the `fruits` table. To retrieve all the data, we will use the `query` method again:

```
Uni<RowSet> rowSet = client.query("SELECT id, name FROM fruits
ORDER BY name ASC");
```

When the operation completes, we will get a `RowSet` that has all the rows buffered in memory. As an `java.lang.Iterable<Row>`, it can be traversed with a *for-each* loop:

```
Uni<List<Fruit>> fruits = rowSet
    .map(pgRowSet -> {
        List<Fruit> list = new ArrayList<>(pgRowSet.size());
        for (Row row : pgRowSet) {
            list.add(from(row));
        }
        return list;
    });
```

The `from` method converts a `Row` instance to a `Fruit` instance. It is extracted as a convenience for the implementation of the other data management methods:

src/main/java/org/acme/vertx/Fruit.java

```java
private static Fruit from(Row row) {
    return new Fruit(row.getLong("id"), row.getString("name"));
}
```

Putting it all together, the `Fruit.findAll` method looks like:

src/main/java/org/acme/vertx/Fruit.java

```java
    public static Uni<List<Fruit>> findAll(PgPool client) {
        return client.query("SELECT id, name FROM fruits ORDER BY name ASC")
            .map(pgRowSet -> {
                List<Fruit> list = new ArrayList<>(pgRowSet.size());
                for (Row row : pgRowSet) {
                    list.add(from(row));
                }
                return list;
            });
    }
```

And the endpoint to get all fruits from the backend:

src/main/java/org/acme/vertx/FruitResource.java

```java
@GET
public Uni<Response> get() {
    return Fruit.findAll(client)
            .map(Response::ok)
            .map(ResponseBuilder::build);
}
```

Now start Quarkus in `dev` mode with:

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

Lastly, open your browser and navigate to http://localhost:8080/fruits, you should see:

```
[{"id":3,"name":"Apple"},{"id":1,"name":"Orange"},{"id":2,"name":"Pear"}]
```

Prepared queries

The Reactive PostgreSQL Client can also prepare queries and take parameters that are replaced in the SQL statement at execution time:

```
client.preparedQuery("SELECT name FROM fruits WHERE id = $1",
Tuple.of(id))
```

> 💡 The SQL string can refer to parameters by position, using $1, $2, …etc.

Like the simple `query` method, `preparedQuery` returns an instance of `Uni<RowSet>`. Equipped with this tooling, we are able to safely use an `id` provided by the user to get the details of a particular fruit:

src/main/java/org/acme/vertx/Fruit.java

```
public static Uni<Fruit> findById(PgPool client, Long id) {
    return client.preparedQuery("SELECT id, name FROM fruits WHERE
id = $1", Tuple.of(id)) ①
            .map(RowSet::iterator) ②
            .map(iterator -> iterator.hasNext() ? from(iterator
.next()) : null); ③
}
```

① Create a `Tuple` to hold the prepared query parameters.

② Get an `Iterator` for the `RowSet` result.

③ Create a `Fruit` instance from the `Row` if an entity was found.

And in the JAX-RS resource:

src/main/java/org/acme/vertx/FruitResource.java

```
@GET
@Path("{id}")
public Uni<Response> getSingle(@PathParam Long id) {
    return Fruit.findById(client, id)
            .map(fruit -> fruit != null ? Response.ok(fruit) :
Response.status(Status.NOT_FOUND)) ①
            .map(ResponseBuilder::build); ②
}
```

① Prepare a JAX-RS response with either the `Fruit` instance if found or the `404` status code.

② Build and send the response.

The same logic applies when saving a `Fruit`:

src/main/java/org/acme/vertx/Fruit.java

```java
public Uni<Long> save(PgPool client) {
    return client.preparedQuery("INSERT INTO fruits (name) VALUES
($1) RETURNING (id)", Tuple.of(name))
            .map(pgRowSet -> pgRowSet.iterator().next().getLong("
id"));
}
```

And in the web resource we handle the POST request:

src/main/java/org/acme/vertx/FruitResource.java

```java
@POST
public Uni<Response> create(Fruit fruit) {
    return fruit.save(client)
            .map(id -> URI.create("/fruits/" + id))
            .map(uri -> Response.created(uri).build());
}
```

Result metadata

A RowSet does not only hold your data in memory, it also gives you some information about the data itself, such as:

- the number of rows affected by the query (inserted/deleted/updated/retrieved depending on the query type),
- the column names.

Let's use this to support removal of fruits in the database:

src/main/java/org/acme/vertx/Fruit.java

```java
public static Uni<Boolean> delete(PgPool client, Long id) {
    return client.preparedQuery("DELETE FROM fruits WHERE id = $1",
Tuple.of(id))
            .map(pgRowSet -> pgRowSet.rowCount() == 1); ①
}
```

① Inspect metadata to determine if a fruit has been actually deleted.

And to handle the HTTP DELETE method in the web resource:

src/main/java/org/acme/vertx/FruitResource.java

```java
@DELETE
@Path("{id}")
public Uni<Response> delete(@PathParam Long id) {
    return Fruit.delete(client, id)
            .map(deleted -> deleted ? Status.NO_CONTENT : Status
.NOT_FOUND)
            .map(status -> Response.status(status).build());
}
```

With GET, POST and DELETE methods implemented, we may now create a minimal web page to try the RESTful application out. We will use jQuery to simplify interactions with the backend:

```html
<!doctype html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>Reactive PostgreSQL Client – Quarkus</title>
    <script src="https://code.jquery.com/jquery-3.3.1.min.js"
            integrity="sha256-
FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=" crossorigin=
"anonymous"></script>
    <script type="application/javascript" src="fruits.js"></script>
</head>
<body>

<h1>Fruits API Testing</h1>

<h2>All fruits</h2>
<div id="all-fruits"></div>

<h2>Create Fruit</h2>
<input id="fruit-name" type="text">
<button id="create-fruit-button" type="button">Create</button>
<div id="create-fruit"></div>

</body>
</html>
```

In the Javascript code, we need a function to refresh the list of fruits when:

- the page is loaded, or

- a fruit is added, or

- a fruit is deleted.

```
function refresh() {
    $.get('/fruits', function (fruits) {
        var list = '';
        (fruits || []).forEach(function (fruit) { ①
            list = list
                + '<tr>'
                + '<td>' + fruit.id + '</td>'
                + '<td>' + fruit.name + '</td>'
                + '<td><a href="#" onclick="deleteFruit(' + fruit
.id + ')">Delete</a></td>'
                + '</tr>'
        });
        if (list.length > 0) {
            list = ''
                +
'<table><thead><th>Id</th><th>Name</th><th></th></thead>'
                + list
                + '</table>';
        } else {
            list = "No fruits in database"
        }
        $('#all-fruits').html(list);
    });
}

function deleteFruit(id) {
    $.ajax('/fruits/' + id, {method: 'DELETE'}).then(refresh);
}

$(document).ready(function () {

    $('#create-fruit-button').click(function () {
        var fruitName = $('#fruit-name').val();
        $.post({
            url: '/fruits',
            contentType: 'application/json',
            data: JSON.stringify({name: fruitName})
        }).then(refresh);
    });

    refresh();
});
```

① The `fruits` parameter is not defined when the database is empty.

All done! Navigate to http://localhost:8080/fruits.html and read/create/delete some fruits.

Database Clients details

Database	Extension name	Pool class name
PostgreSQL	`quarkus-reactive-pg-client`	`io.vertx.mutiny.pgclient.PgPool`
MySQL	`quarkus-reactive-mysql-client`	`io.vertx.mutiny.mysqlclient.MySQLPool`

Configuration Reference

MySQL

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

Configuration property	Type	Default
`quarkus.datasource.reactive.mysql.cache-prepared-statements` Whether prepared statements should be cached on the client side.	boolean	
`quarkus.datasource.reactive.mysql.charset` Charset for connections.	string	
`quarkus.datasource.reactive.mysql.collation` Collation for connections.	string	
`quarkus.reactive-mysql-client.cache-prepared-statements`	boolean	
`quarkus.reactive-mysql-client.charset`	string	
`quarkus.reactive-mysql-client.collation`	string	

PostgreSQL

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

Configuration property	Type	Default

`quarkus.reactive-pg-client.cache-prepared-statements`	boolean	
`quarkus.reactive-pg-client.pipelining-limit`	int	
`quarkus.datasource.reactive.postgresql.cache-prepared-statements` Whether prepared statements should be cached on the client side.	boolean	
`quarkus.datasource.reactive.postgresql.pipelining-limit` The maximum number of inflight database commands that can be pipelined.	int	