Quarkus - Getting started with Reactive

Learn how to create a reactive application with Quarkus and explore the different reactive features offered by Quarkus. This guide covers:

- A quick glance at the Quarkus engine and how it enables reactive
- A brief introduction to Mutiny - the reactive programming library used by Quarkus
- Bootstrapping a reactive application
- Creating a reactive JAX-RS endpoint (asynchronous, streams…)
- Using reactive database access
- Interacting with other reactive APIs

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 8 or 11+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+

Solutions

We recommend that you follow the instructions from Boostrapping project and onwards to create the application step by step.

However, you can go right to the completed example.

Download an archive or clone the git repository:

```
git clone https://github.com/quarkusio/quarkus-quickstarts.git
```

The solutions are located in the `getting-started-reactive` and `getting-started-reactive-crud` directories.

The multiple reactive facets of Quarkus

Quarkus is reactive. If you look under the hood, you will find a reactive engine powering your Quarkus application. This engine is Eclipse Vert.x (https://vertx.io). Every IO interaction passes through the

non-blocking and reactive Vert.x engine.

Let's take 2 examples to explain how it works. Imagine an incoming HTTP request. The (Vert.x) HTTP server receives the request and then routes it to the application. If the request targets a JAX-RS resource, the routing layer invokes the resource method in a worker thread and writes the response when the data is available. So far, nothing new or outstanding. The following picture depicts this behavior.

But if the HTTP request targets a reactive (non-blocking) route, the routing layer invokes the route on the IO thread giving lots of benefits such as higher concurrency and performance:

As a consequence, many Quarkus components are designed with reactive in mind, such as database access (PostgreSQL, MySQL, Mongo, etc.), application services (mail, template engine, etc.), messaging (Kafka, AMQP, etc.) and so on. But, to fully benefit from this model, the application code should be written in a non-blocking manner. That's where having a reactive API is an ultimate weapon.

Mutiny - A reactive programming library

Mutiny is a reactive programming library allowing to express and compose asynchronous actions. It offers 2 types:

- `io.smallrye.mutiny.Uni` - for asynchronous action providing 0 or 1 result

- `io.smallrye.mutiny.Multi` - for multi-item (with back-pressure) streams

Both types are lazy and follow a subscription pattern. The computation only starts once there is an actual need for it (i.e. a subscriber enlists).

```
uni.subscribe().with(
    result -> System.out.println("result is " + result),
    failure -> failure.printStackTrace()
);

multi.subscribe().with(
    item -> System.out.println("Got " + item),
    failure -> failure.printStackTrace()
);
```

Both `Uni` and `Multi` expose event-driven APIs: you express what you want to do upon a given event

(success, failure, etc.). These APIs are divided into groups (types of operations) to make it more expressive and avoid having 100s of methods attached to a single class. The main types of operations are about reacting to failure, completion, manipulating items, extracting, or collecting them. It provides a smooth coding experience, with a navigable API, and the result does not require too much knowledge around reactive.

```
httpCall
    .onFailure().recoverWithItem("my fallback");
```

You may wonder about Reactive Streams (https://www.reactive-streams.org/). `Multi` implements Reactive Streams `Publisher`, and so implements the Reactive Streams back-pressure mechanism. `Uni` does not implement `Publisher` as the subscription to the `Uni` is enough to indicate you are interested in the result. It is again with the idea of simpler and smoother APIs in mind as the Reactive Streams subscription/request ceremony is more complex.

Embracing the unification of reactive and imperative pillars from Quarkus, both `Uni` and `Multi` provide bridges to imperative constructs. For example, you can transform a `Multi` into an `Iterable` or *await* the item produced by a `Uni`.

```
// Block until the result is available
String result = uni.await().indefinitely();

// Transform an asynchronous stream into a blocking iterable
stream.subscribe().asIterable().forEach(s -> System.out.println(
"Item is " + s));
```

At that point, if you are a RxJava or a Reactor user, you may wonder how you can use your familiar `Flowable`, `Single`, `Flux`, `Mono`... Mutiny allows converting `Unis` and `Multis` from and to RX Java and Reactor types:

```
Maybe<String> maybe = uni.convert().with(UniRxConverters.toMaybe()
);
Flux<String> flux = multi.convert().with(MultiReactorConverters
.toFlux());
```

But, what about Vert.x? Vert.x APIs are also available using Mutiny types. The following snippet shows a usage of the Vert.x Web Client:

```java
// Use io.vertx.mutiny.ext.web.client.WebClient
client = WebClient.create(vertx,
                new WebClientOptions().setDefaultHost(
"fruityvice.com").setDefaultPort(443).setSsl(true)
                        .setTrustAll(true));
// ...
Uni<JsonObject> uni =
    client.get("/api/fruit/" + name)
        .send()
        .onItem().apply(resp -> {
            if (resp.statusCode() == 200) {
                return resp.bodyAsJsonObject();
            } else {
                return new JsonObject()
                        .put("code", resp.statusCode())
                        .put("message", resp.bodyAsString());
            }
        });
```

Last but not least, Mutiny has built-in integration with MicroProfile Context Propagation so you can propagate transactions, traceability data, and so on in your reactive pipeline.

But enough talking, let's get our hands dirty!

Bootstrapping the project

The easiest way to create a new Quarkus project is to open a terminal and run the following command:

For Linux and macOS users

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=getting-started-reactive \
    -DclassName="org.acme.quickstart.ReactiveGreetingResource" \
    -Dpath="/hello" \
    -Dextensions="resteasy-mutiny, resteasy-jsonb"
cd getting-started-reactive
```

For Windows users

- If using cmd, (don't use forward slash \)

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR1:create
-DprojectGroupId=org.acme -DprojectArtifactId=getting-started
-reactive
-DclassName="org.acme.quickstart.ReactiveGreetingResource"
-Dpath="/hello" -Dextensions="resteasy-mutiny,resteasy-jsonb"
```

- If using Powershell, wrap -D parameters in double quotes

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR1:create "-
DprojectGroupId=org.acme" "-DprojectArtifactId=getting-started-
reactive" "-
DclassName=org.acme.quickstart.ReactiveGreetingResource" "-
Dpath=/hello" "-Dextensions=resteasy-mutiny,resteasy-jsonb"
```

It generates the following in `./getting-started-reactive`:

- the Maven structure

- an `org.acme.quickstart.ReactiveGreetingResource` resource exposed on `/hello`

- an associated unit test

- a landing page that is accessible on `http://localhost:8080` after starting the application

- example `Dockerfile` files for both `native` and `jvm` modes in `src/main/docker`

- the application configuration file

The generated `pom.xml` also declares the RESTEasy Mutiny support and RESTEasy JSON-B to serialize payloads.

Reactive JAX-RS resources

During the project creation, the `src/main/java/org/acme/quickstart/ReactiveGreetingResource.java` file has been created with the following content:

```java
package org.acme.quickstart;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class ReactiveGreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

It's a very simple REST endpoint, returning "hello" to requests on "/hello".

Let's now create a `ReactiveGreetingService` class with the following content:

```java
package org.acme.getting.started;

import io.smallrye.mutiny.Multi;
import io.smallrye.mutiny.Uni;

import javax.enterprise.context.ApplicationScoped;
import java.time.Duration;

@ApplicationScoped
public class ReactiveGreetingService {

    public Uni<String> greeting(String name) {
        return Uni.createFrom().item(name)
                .onItem().apply(n -> String.format("hello %s",
name));
    }
}
```

Then, edit the `ReactiveGreetingResource` class to match the following content:

```java
package org.acme.getting.started;

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 io.smallrye.mutiny.Multi;
import io.smallrye.mutiny.Uni;
import org.jboss.resteasy.annotations.SseElementType;
import org.jboss.resteasy.annotations.jaxrs.PathParam;
import org.reactivestreams.Publisher;

@Path("/hello")
public class ReactiveGreetingResource {

    @Inject
    ReactiveGreetingService service;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("/greeting/{name}")
    public Uni<String> greeting(@PathParam String name) {
        return service.greeting(name);
    }

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

The `ReactiveGreetingService` class contains a straightforward method producing a `Uni`. While, in this example, the resulting item is emitted immediately, you can imagine any async API producing a `Uni`. We cover this later in this guide.

Now, start the application using:

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

Once running, check that you get the expected greeting message by opening http://localhost:8080/hello/greeting/neo.

Handling streams

So far, we only return an asynchronous result. In this section, we extend the application with streams conveying multiple items. These streams could come from Kafka or any other source of data, but to keep things simple, we just generate periodic greeting messages.

In the ReactiveGreetingService, add the following method:

```
public Multi<String> greetings(int count, String name) {
  return Multi.createFrom().ticks().every(Duration.ofSeconds(1))
        .onItem().apply(n -> String.format("hello %s - %d", name,
n))
        .transform().byTakingFirstItems(count);
}
```

It generates a greeting message every second and stops after count messages.

In the ReactiveGreetingResource add the following method:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/greeting/{count}/{name}")
public Multi<String> greetings(@PathParam int count, @PathParam
String name) {
  return service.greetings(count, name);
}
```

This endpoint streams the items to the client as a JSON Array. The name and number of messages are parameterized using path parameters.

So calling the endpoint produces something like:

```
$ curl http://localhost:8080/hello/greeting/3/neo
["hello neo - 0","hello neo - 1","hello neo - 2"]
```

We can also generate Server-Sent Event responses by returning a Multi:

```java
@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
@SseElementType(MediaType.TEXT_PLAIN)
@Path("/stream/{count}/{name}")
public Multi<String> greetingsAsStream(@PathParam int count,
@PathParam String name) {
    return service.greetings(count, name);
}
```

The only difference with the previous snippet is the produced type and the `@SseElementType` annotation indicating the type of each event. As the `@Produces` annotation defines `SERVER_SENT_EVENTS`, JAX-RS needs it to knows the content type of each (nested) event.

You can see the result using:

```
$ curl -N http://localhost:8080/hello/stream/5/neo
data: hello neo - 0

data: hello neo - 1

data: hello neo - 2

data: hello neo - 3

data: hello neo - 4
```

Using Reactive APIs

Using Quarkus reactive APIs

Quarkus provides many reactive APIs using the Mutiny model. In this section, we are going to see how you can use the Reactive PostgreSQL driver to interact with your database in a non-blocking and reactive way.

Create a new project using:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=getting-started-reactive-crud \
    -DclassName="org.acme.reactive.crud.FruitResource" \
    -Dpath="/fruits" \
    -Dextensions="resteasy-mutiny, resteasy-jsonb, reactive-pg-
client"
cd getting-started-reactive-crud
```

This application is interacting with a PostgreSQL database, so you need one:

```
docker run --ulimit memlock=-1:-1 -it --rm=true --memory
-swappiness=0 \
           --name postgres-quarkus-reactive -e
POSTGRES_USER=quarkus_test \
           -e POSTGRES_PASSWORD=quarkus_test -e
POSTGRES_DB=quarkus_test \
           -p 5432:5432 postgres:11.2
```

Then, let's configure our datasource. Open the src/main/resources/application.properties and add the following content:

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=quarkus_test
quarkus.datasource.password=quarkus_test
quarkus.datasource.reactive.url=postgresql://localhost:5432/quarkus
_test
myapp.schema.create=true
```

The 3 first lines define the datasource. The last line is going to be used in the application to indicate whether we insert a few items when the application gets initialized.

Now, let's create our *entity*. Create the org.acme.reactive.crud.Fruit class with the following content:

```java
package org.acme.reactive.crud;

import io.smallrye.mutiny.Multi;
import io.smallrye.mutiny.Uni;
import io.vertx.mutiny.pgclient.PgPool;
import io.vertx.mutiny.sqlclient.Row;
import io.vertx.mutiny.sqlclient.RowSet;
import io.vertx.mutiny.sqlclient.Tuple;

import java.util.stream.StreamSupport;

public class Fruit {

    public Long id;

    public String name;

    public Fruit() {
        // default constructor.
    }
```

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

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

    public static Multi<Fruit> findAll(PgPool client) {
        return client.query("SELECT id, name FROM fruits ORDER BY
name ASC")
                // Create a Multi from the set of rows:
                .onItem().produceMulti(set -> Multi.createFrom()
.items(() -> StreamSupport.stream(set.spliterator(), false)))
                // For each row create a fruit instance
                .onItem().apply(Fruit::from);
    }

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

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

    public Uni<Boolean> update(PgPool client) {
        return client.preparedQuery("UPDATE fruits SET name = $1
WHERE id = $2", Tuple.of(name, id))
                .onItem().apply(pgRowSet -> pgRowSet.rowCount() ==
1);
    }

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

    private static Fruit from(Row row) {
```

```
        return new Fruit(row.getLong("id"), row.getString("name"));
    }
}
```

This *entity* contains a few fields and methods to find, update, and delete rows from the database. These methods return either `Unis` or `Multis` as the produced items are emitted asynchronously when the results have been retrieved. Notice that the reactive PostgreSQL client already provides `Uni` and `Multi` instances. So you only transform the results from the database into *business-friendly* objects.

Then, let's use this `Fruit` class in the `FruitResource`. Edit the `FruitResource` class to match the following content:

```
package org.acme.reactive.crud;

import io.smallrye.mutiny.Multi;
import io.smallrye.mutiny.Uni;
import io.vertx.mutiny.pgclient.PgPool;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;
import javax.ws.rs.core.Response.Status;
import java.net.URI;

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

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

    @Inject
    PgPool client;
```

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

    private void initdb() {
        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 ('Kiwi')"))
                .flatMap(r -> client.query("INSERT INTO fruits
(name) VALUES ('Durian')"))
                .flatMap(r -> client.query("INSERT INTO fruits
(name) VALUES ('Pomelo')"))
                .flatMap(r -> client.query("INSERT INTO fruits
(name) VALUES ('Lychee')"))
                .await().indefinitely();
    }

    @GET
    public Multi<Fruit> get() {
        return Fruit.findAll(client);
    }

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

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

    @PUT
    @Path("{id}")
    public Uni<Response> update(@PathParam Long id, Fruit fruit) {
        return fruit.update(client)
                .onItem().apply(updated -> updated ? Status.OK :
```

```
Status.NOT_FOUND)
                .onItem().apply(status -> Response.status(status)
.build());
    }

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

This resource returns `Uni` and `Multi` instances based on the result produced by the `Fruit` class.

Using Vert.x clients

The previous example uses a *service* provided by Quarkus. Also, you can use Vert.x clients directly. There is a Mutiny version of the Vert.x APIs. This API is divided into several artifacts you can import independently:

groupId:artifactId	Description
`io.smallrye.reactive:smallrye-mutiny-vertx-core`	Mutiny API for Vert.x Core
`io.smallrye.reactive:smallrye-mutiny-vertx-mail-client`	Mutiny API for the Vert.x Mail Client
`io.smallrye.reactive:smallrye-mutiny-vertx-web-client`	Mutiny API for the Vert.x Web Client
`io.smallrye.reactive:smallrye-mutiny-vertx-mongo-client`	Mutiny API for the Vert.x Mongo Client
`io.smallrye.reactive:smallrye-mutiny-vertx-redis-client`	Mutiny API for the Vert.x Redis Client
`io.smallrye.reactive:smallrye-mutiny-vertx-cassandra-client`	Mutiny API for the Vert.x Cassandra Client
`io.smallrye.reactive:smallrye-mutiny-vertx-consul-client`	Mutiny API for the Vert.x Consul Client
`io.smallrye.reactive:smallrye-mutiny-vertx-kafka-client`	Mutiny API for the Vert.x Kafka Client
`io.smallrye.reactive:smallrye-mutiny-vertx-amqp-client`	Mutiny API for the Vert.x AMQP Client

groupId:artifactId	Description
`io.smallrye.reactive:smallrye-` `mutiny-vertx-rabbitmq-client`	Mutiny API for the Vert.x RabbitMQ Client

You can also check the available APIs on http://smallrye.io/smallrye-reactive-utils/apidocs/.

Let's take an example. Add the following dependency to your application:

```xml
<dependency>
    <groupId>io.smallrye.reactive</groupId>
    <artifactId>smallrye-mutiny-vertx-web-client</artifactId>
</dependency>
```

It provides the Mutiny API of the Vert.x Web Client. Then, you can use the web client as follows:

```java
package org.acme.vertx;

import io.smallrye.mutiny.Uni;
import io.vertx.core.json.JsonObject;
import io.vertx.ext.web.client.WebClientOptions;
import io.vertx.mutiny.core.Vertx;
import io.vertx.mutiny.ext.web.client.WebClient;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

import javax.annotation.PostConstruct;
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;

@Path("/fruit-data")
public class ResourceUsingWebClient {

    @Inject
    Vertx vertx;

    private WebClient client;

    @PostConstruct
    void initialize() {
        this.client = WebClient.create(vertx,
                new WebClientOptions().setDefaultHost(
 "fruityvice.com").setDefaultPort(443).setSsl(true)
                        .setTrustAll(true));
    }
```

```java
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{name}")
    public Uni<JsonObject> getFruitData(@PathParam("name") String
 name) {
        return client.get("/api/fruit/" + name)
                .send()
                .map(resp -> {
                    if (resp.statusCode() == 200) {
                        return resp.bodyAsJsonObject();
                    } else {
                        return new JsonObject()
                                .put("code", resp.statusCode())
                                .put("message", resp.bodyAsString(
 ));
                    }
                });
    }

 }
```

There are 2 important points:

1. The injected Vert.x instance has the `io.vertx.mutiny.core.Vertx` type which is the Mutiny variant of Vert.x;

2. The Web Client is created from `io.vertx.mutiny.ext.web.client.WebClient`.

The Mutiny version of the Vert.x APIs also offers:

- `andAwait` methods such as `sendAndAwait`. `andAwait` indicates that the caller thread is blocked until the result is available. Be aware not to block the event loop / IO thread that way.

- `andForget` methods such as `writeAndForget`. `andForget` is available for method returning a `Uni`. `andForget` indicates that you don't need the resulting `Uni` indicating the success or failure of the operation. However, remember that if you don't subscribe, the operation would not be triggered. `andForget` manages this for you and manage the subscription.

- `toMulti` methods allowing to transform a Vert.x `ReadStream` into a `Multi`

- `toBlockingIterable` / `toBlockingStream` methods allowing to transform a Vert.x `ReadStream` into a blocking iterable or blocking `java.util.Stream`

Using RxJava or Reactor APIs

Mutiny provides utilities to convert RxJava 2 and Project Reactor types to `Uni` and `Multi`.

RxJava 2 converters are available in the following dependency:

```
<dependency>
    <groupId>io.smallrye.reactive</groupId>
    <artifactId>mutiny-rxjava</artifactId>
</dependency>
```

So if you have an API returning RxJava 2 types (Completable, Single, Maybe, Observable, Flowable), you can create Unis and Multis as follows:

```java
import io.smallrye.mutiny.converters.multi.MultiRxConverters;
import io.smallrye.mutiny.converters.uni.UniRxConverters;
// ...
Uni<Void> uniFromCompletable = Uni.createFrom().converter
(UniRxConverters.fromCompletable(), completable);
Uni<String> uniFromSingle = Uni.createFrom().converter
(UniRxConverters.fromSingle(), single);
Uni<String> uniFromMaybe = Uni.createFrom().converter
(UniRxConverters.fromMaybe(), maybe);
Uni<String> uniFromEmptyMaybe = Uni.createFrom().converter
(UniRxConverters.fromMaybe(), emptyMaybe);
Uni<String> uniFromObservable = Uni.createFrom().converter
(UniRxConverters.fromObservable(), observable);
Uni<String> uniFromFlowable = Uni.createFrom().converter
(UniRxConverters.fromFlowable(), flowable);

Multi<Void> multiFromCompletable = Multi.createFrom().converter
(MultiRxConverters.fromCompletable(), completable);
Multi<String> multiFromSingle = Multi.createFrom().converter
(MultiRxConverters.fromSingle(), single);
Multi<String> multiFromMaybe = Multi.createFrom().converter
(MultiRxConverters.fromMaybe(), maybe);
Multi<String> multiFromEmptyMaybe = Multi.createFrom().converter
(MultiRxConverters.fromMaybe(), emptyMaybe);
Multi<String> multiFromObservable = Multi.createFrom().converter
(MultiRxConverters.fromObservable(), observable);
Multi<String> multiFromFlowable = Multi.createFrom().converter
(MultiRxConverters.fromFlowable(), flowable);
```

You can also transform Unis and Multis into RxJava types:

```
Completable completable = uni.convert().with(UniRxConverters
.toCompletable());
Single<Optional<String>> single = uni.convert().with
(UniRxConverters.toSingle());
Single<String> single2 = uni.convert().with(UniRxConverters
.toSingle().failOnNull());
Maybe<String> maybe = uni.convert().with(UniRxConverters.toMaybe()
);
Observable<String> observable = uni.convert().with(UniRxConverters
.toObservable());
Flowable<String> flowable = uni.convert().with(UniRxConverters
.toFlowable());
// ...
Completable completable = multi.convert().with(MultiRxConverters
.toCompletable());
Single<Optional<String>> single = multi.convert().with
(MultiRxConverters.toSingle());
Single<String> single2 = multi.convert().with(MultiRxConverters
        .toSingle().onEmptyThrow(() -> new Exception("D'oh!")));
Maybe<String> maybe = multi.convert().with(MultiRxConverters
.toMaybe());
Observable<String> observable = multi.convert().with
(MultiRxConverters.toObservable());
Flowable<String> flowable = multi.convert().with(MultiRxConverters
.toFlowable());
```

Project Reactor converters are available in the following dependency:

```
<dependency>
    <groupId>io.smallrye.reactive</groupId>
    <artifactId>mutiny-reactor</artifactId>
</dependency>
```

So if you have an API returning Reactor types (Mono, Flux), you can create Unis and Multis as follows:

```
import io.smallrye.mutiny.converters.multi.MultiReactorConverters;
import io.smallrye.mutiny.converters.uni.UniReactorConverters;
// ...
Uni<String> uniFromMono = Uni.createFrom().converter
(UniReactorConverters.fromMono(), mono);
Uni<String> uniFromFlux = Uni.createFrom().converter
(UniReactorConverters.fromFlux(), flux);

Multi<String> multiFromMono = Multi.createFrom().converter
(MultiReactorConverters.fromMono(), mono);
Multi<String> multiFromFlux = Multi.createFrom().converter
(MultiReactorConverters.fromFlux(), flux);
```

You can also transform `Unis` and `Multis` into Reactor types:

```
Mono<String> mono = uni.convert().with(UniReactorConverters.toMono
());
Flux<String> flux = uni.convert().with(UniReactorConverters.toFlux
());

Mono<String> mono2 = multi.convert().with(MultiReactorConverters
.toMono());
Flux<String> flux2 = multi.convert().with(MultiReactorConverters
.toFlux());
```

Using CompletionStages or Publisher API

If you are facing an API using `CompletionStage`, `CompletableFuture`, or `Publisher`, you can convert back and forth. First, both `Uni` and `Multi` can be created from a `CompletionStage` or from a `Supplier<CompletionStage>`. For example:

```
CompletableFuture<String> future = Uni
        // Create from a Completion Stage
        .createFrom().completionStage(CompletableFuture.
supplyAsync(() -> "hello"));
```

On `Uni`, you can also produce a `CompletionStage` using `subscribeAsCompletionStage()` that produces a `CompletionStage` that would get the item or failure emitted by the `Uni`.

You can also create `Unis` and `Multis` from instances of `Publisher` using `createFrom().publisher(Publisher)`. You can transform a `Uni` into a `Publisher` using `toMulti`. Indeed, `Multi` implements `Publisher`.

What's next?

This guide is an introduction to reactive in Quarkus. There are plenty of Quarkus features that are already reactive. The following list gives you a few examples:

- Sending email

- Using MongoDB and MongoDB with Panache

- Reactive Database Clients

- Using Vert.x

- Interacting with Kafka and Interacting with AMQP

- Using Neo4J

- Using reactive routes