

Lunatech Beginner Quarkus Course

v1.0

Course Overview

Welcome to Lunatech Beginner Quarkus

In this course you will learn:

- Quarkus Philosophy
- Dependency Injection with ArC
- Quarkus Data Access Layer
- The Qute Template Engine
- RESTful API development
- Reactive Programming
- Observability
- Testing
- ... and a lot of Quarkus tooling and best practices

Prerequisites

You should have:

- Knowledge of Java
- Basic knowledge of the Web and HTTP

Your machine should have:

- Java 11
- Docker
- An IDE of your choice

Agenda day 1

Morning

- Getting started
- Qute Template Engine
- Database access

Afternoon

- Web Services
- OpenAPI
- Testing

Agenda day 2

Morning

- Quarkus execution model
- Reactive programming
- Reactive messaging

Afternoon

- Reactive programming (cont'd)
- Optional material (depending on time)
 - Observability
 - Serverless with AWS

Case Study

We will be doing several exercises that build upon the same domain: a furniture store called HIQUÉA.

About the trainer(s)

About the participants

Practicalities

- Daily schedule: 09:30 - 17:30
- Student GitHub Repo

Getting Started

Learning outcomes

After this module, you should:

- Understand the Quarkus philosophy and its motivations
- Witness how to generate a new Quarkus project from the Quarkus website
- Know how to run Quarkus from the terminal with the Maven wrapper script
- Know how to access the Dev UI

Why Quarkus exists

Supersonic Subatomic Java ... A Kubernetes Native Java stack... crafted from the best of breed Java libraries and standards.

<https://quarkus.io/>

Why Quarkus exists

- *Java for the cloud-native age*
- Unifies Imperative & Reactive paradigms
- Developer Joy

Why Quarkus exists

- Java for the cloud-native age
- *Unifies Imperative & Reactive paradigms*
- Developer Joy

Why Quarkus exists

- Java for the cloud-native age
- Unifies Imperative & Reactive paradigms
- *Developer Joy*

Based on standards, inspired by best practice

- Quarkus and many extensions are based on industry standards like Jakarta EE and MicroProfile
- Quarkus itself is built on a best-of-breed reactive framework Vert.x
- Quarkus inspired by developer experience of other frameworks like Spring Boot and Play Java
 - Quarkus even has a *Spring Compatibility* extension
- Quarkus implements many APIs and supports a lot of frameworks. This is part of the Quarkus Philosophy.

Hello World Demo

Hello World in Quarkus

We can use JAX-RS annotations to create a Hello World endpoint:

```
package com.lunatech.training.quarkus;

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 HelloResource {

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

JAX-RS & RESTeasy

JAX-RS is a Jakarta EE API spec. It contains annotations such as:

- `@Path`
- `@GET, @PUT, @POST, @DELETE, @HEAD`
- `@Produces` and `@Consumes`
- `@PathParam, @QueryParam, @HeaderParam` and more...

So, a standard way of describing RESTful web services

JAX-RS & RESTeasy

RESTEasy is the Red Hat *implementation* of the JAX-RS standard. It's what Quarkus uses to provide web services.

Exercise: Hello World

Recap

In this module we have:

- Discussed the philosophy of Quarkus
- Set up the base project for the rest of the training
- Experienced some of the Developer Joy that Quarkus aims to spark

Qute

Learning outcomes

After this module, you should:

- Understand the motivations behind Qute
- Know how to create and use a simple Qute template
- Know how to create custom Qute tags and Extension methods

Qute Template Engine

Quarkus comes with a template engine named *Qute* (**Quarkus templating**):

- Simple syntax
- Minimized reflection usage
- Optionally type-safe
- Output can be streamed

Detour: Quarkus Extensions

- Can think of them as project dependencies, but with added dimensions
 - Build-time augmentation
- They help third-party libraries integrate more easily into Quarkus applications and build
- Integrations can more easily target GraalVM
- Statuses: Stable, Preview, Experimental

Qute Expressions

Given a class Product:

```
public class Product {  
    public String name;  
    public BigDecimal price;  
}
```

This is a template rendering the product details:

```
<html>  
    <head>  
        <title>{product.name}</title>  
    </head>  
    <body>  
        <h1>{product.name}</h1>  
        <div>Price: {product.price}</div>  
    </body>  
</html>
```

Quite iterations

You can iterate over collections:

```
<ul>
{#for product in products}
  <li>{product.name}</li>
{/for}
</ul>
```

Some Qute operators

Qute has some useful operators:

```
1 Manufacturer: {product.manufacturer ?: 'Unknown' }
2 Manufacturer: {product.manufacturer or 'Unknown' }
3 Available: {product.isAvailable ? 'Yep' : 'Nope' }
4
5 {product.isAvailable && product.isCool}
6 {product.isAvailable || product.isCool}
```

Some Qute operators

Qute has some useful operators:

```
1 Manufacturer: {product.manufacturer ?: 'Unknown' }
2 Manufacturer: {product.manufacturer or 'Unknown' }
3 Available: {product.isAvailable ? 'Yep' : 'Nope' }
4
5 {product.isAvailable && product.isCool}
6 {product.isAvailable || product.isCool}
```

Some Qute operators

Qute has some useful operators:

```
1 Manufacturer: {product.manufacturer ?: 'Unknown' }
2 Manufacturer: {product.manufacturer or 'Unknown' }
3 Available: {product.isAvailable ? 'Yep' : 'Nope' }
4
5 {product.isAvailable && product.isCool}
6 {product.isAvailable || product.isCool}
```

Some Qute operators

Qute has some useful operators:

```
1 Manufacturer: {product.manufacturer ?: 'Unknown' }
2 Manufacturer: {product.manufacturer or 'Unknown' }
3 Available: {product.isAvailable ? 'Yep' : 'Nope' }
4
5 {product.isAvailable && product.isCool}
6 {product.isAvailable || product.isCool}
```

Some Qute operators

Qute has some useful operators:

```
1 Manufacturer: {product.manufacturer ?: 'Unknown' }
2 Manufacturer: {product.manufacturer or 'Unknown' }
3 Available: {product.isAvailable ? 'Yep' : 'Nope' }
4
5 {product.isAvailable && product.isCool}
6 {product.isAvailable || product.isCool}
```

Some Qute operators

Qute has some useful operators:

```
1 Manufacturer: {product.manufacturer ?: 'Unknown' }
2 Manufacturer: {product.manufacturer or 'Unknown' }
3 Available: {product.isAvailable ? 'Yep' : 'Nope' }
4
5 {product.isAvailable && product.isCool}
6 {product.isAvailable || product.isCool}
```

Some Qute operators

Qute has some useful operators:

```
1 Manufacturer: {product.manufacturer ?: 'Unknown' }
2 Manufacturer: {product.manufacturer or 'Unknown' }
3 Available: {product.isAvailable ? 'Yep' : 'Nope' }
4
5 {product.isAvailable && product.isCool}
6 {product.isAvailable || product.isCool}
```

Qute Usage

```
1 @Inject
2 Template productDetails;
3
4 @GET
5 @Path("{productId}")
6 public TemplateInstance product(@PathParam("productId") long productId) {
7     Product product = Product.findById(productId);
8     return productDetails.data("product", product);
9 }
```

1. Inject a template. Quarkus derives the template file name from the field name.
2. Resource method returns a `TemplateInstance`. RESTeasy knows how to convert this to a response.
3. Populate the template with data to create a `TemplateInstance`.

Qute Usage

```
1 @Inject
2 Template productDetails;
3
4 @GET
5 @Path("{productId}")
6 public TemplateInstance product(@PathParam("productId") long productId) {
7     Product product = Product.findById(productId);
8     return productDetails.data("product", product);
9 }
```

1. Inject a template. Quarkus derives the template file name from the field name.
2. Resource method returns a `TemplateInstance`. RESTeasy knows how to convert this to a response.
3. Populate the template with data to create a `TemplateInstance`.

Qute Usage

```
1 @Inject
2 Template productDetails;
3
4 @GET
5 @Path("{productId}")
6 public TemplateInstance product(@PathParam("productId") long productId) {
7     Product product = Product.findById(productId);
8     return productDetails.data("product", product);
9 }
```

1. Inject a template. Quarkus derives the template file name from the field name.
2. Resource method returns a `TemplateInstance`. RESTeasy knows how to convert this to a response.
3. Populate the template with data to create a `TemplateInstance`.

Qute Usage

```
1 @Inject
2 Template productDetails;
3
4 @GET
5 @Path("{productId}")
6 public TemplateInstance product(@PathParam("productId") long productId) {
7     Product product = Product.findById(productId);
8     return productDetails.data("product", product);
9 }
```

1. Inject a template. Quarkus derives the template file name from the field name.
2. Resource method returns a `TemplateInstance`. RESTeasy knows how to convert this to a response.
3. Populate the template with data to create a `TemplateInstance`.

Qute Usage

```
1 @Inject
2 Template productDetails;
3
4 @GET
5 @Path("{productId}")
6 public TemplateInstance product(@PathParam("productId") long productId) {
7     Product product = Product.findById(productId);
8     return productDetails.data("product", product);
9 }
```

1. Inject a template. Quarkus derives the template file name from the field name.
2. Resource method returns a `TemplateInstance`. RESTeasy knows how to convert this to a response.
3. Populate the template with data to create a `TemplateInstance`.

Exercise: A Quite Hello World

Qute Virtual Methods

Qute allows you to call *virtual methods* on values. They are called *virtual* because they don't correspond to real methods on the Java value

```
1 <p>Name: {name}</p>
2 <p>Name: {name.toUpperCase()}</p>
3 <p>Name: {name.toUpperCase}</p>
```

Qute Virtual Methods

Qute allows you to call *virtual methods* on values. They are called *virtual* because they don't correspond to real methods on the Java value

```
1 <p>Name: {name}</p>
2 <p>Name: {name.toUpperCase()}</p>
3 <p>Name: {name.toUpperCase}</p>
```

Qute Virtual Methods

Qute allows you to call *virtual methods* on values. They are called *virtual* because they don't correspond to real methods on the Java value

```
1 <p>Name: {name}</p>
2 <p>Name: {name.toUpperCase()}</p>
3 <p>Name: {name.toUpperCase}</p>
```

Qute Virtual Methods

```
1 @TemplateExtension
2 public class StringExtension {
3     public static String shout(String in) {
4         return in + "!";
5     }
6
7     public static String shout(String in, String append) {
8         return in + append;
9     }
10 }
```

```
1 <p>Name: {name.shout}</p>
2 <p>Name: {name.shout('!!!!')}</p>
3 <p>Name: {name shout '!!!!'}</p>
```

Qute Virtual Methods

```
1 @TemplateExtension
2 public class StringExtension {
3     public static String shout(String in) {
4         return in + "!";
5     }
6
7     public static String shout(String in, String append) {
8         return in + append;
9     }
10 }
```

```
1 <p>Name: {name.shout}</p>
2 <p>Name: {name.shout('!!!!')}</p>
3 <p>Name: {name shout '!!!!'}</p>
```

Qute Virtual Methods

```
1 @TemplateExtension
2 public class StringExtension {
3     public static String shout(String in) {
4         return in + "!";
5     }
6
7     public static String shout(String in, String append) {
8         return in + append;
9     }
10 }
```

```
1 <p>Name: {name.shout}</p>
2 <p>Name: {name.shout('!!!!')}</p>
3 <p>Name: {name shout '!!!!'}</p>
```

Qute Virtual Methods

```
1 @TemplateExtension
2 public class StringExtension {
3     public static String shout(String in) {
4         return in + "!";
5     }
6
7     public static String shout(String in, String append) {
8         return in + append;
9     }
10 }
```

```
1 <p>Name: {name.shout}</p>
2 <p>Name: {name.shout('!!!!')}</p>
3 <p>Name: {name shout '!!!!'}</p>
```

Qute Virtual Methods

```
1 @TemplateExtension
2 public class StringExtension {
3     public static String shout(String in) {
4         return in + "!";
5     }
6
7     public static String shout(String in, String append) {
8         return in + append;
9     }
10 }
```

```
1 <p>Name: {name.shout}</p>
2 <p>Name: {name.shout('!!!!')}</p>
3 <p>Name: {name shout '!!!!'}</p>
```

Qute Virtual Methods

```
1 @TemplateExtension
2 public class StringExtension {
3     public static String shout(String in) {
4         return in + "!";
5     }
6
7     public static String shout(String in, String append) {
8         return in + append;
9     }
10 }
```

```
1 <p>Name: {name.shout}</p>
2 <p>Name: {name.shout('!!!!')}</p>
3 <p>Name: {name shout '!!!!'}</p>
```

Qute Virtual Methods

```
1 @TemplateExtension
2 public class StringExtension {
3     public static String shout(String in) {
4         return in + "!";
5     }
6
7     public static String shout(String in, String append) {
8         return in + append;
9     }
10 }
```

```
1 <p>Name: {name.shout}</p>
2 <p>Name: {name.shout('!!!!')}</p>
3 <p>Name: {name shout '!!!!'}</p>
```

Qute Virtual Methods

```
1 @TemplateExtension
2 public class StringExtension {
3     public static String shout(String in) {
4         return in + "!";
5     }
6
7     public static String shout(String in, String append) {
8         return in + append;
9     }
10 }
```

```
1 <p>Name: {name.shout}</p>
2 <p>Name: {name.shout('!!!!')}</p>
3 <p>Name: {name shout '!!!!'}</p>
```

Quite Virtual Methods

We can't call *real* methods with parameters out of the box:

```
<p>Name: {name.replace('k', 'c')}</p>
```

Will print:

```
<p>Name: NOT_FOUND</p>
```

Qute Virtual Methods - Template Data

But we can instruct Qute to generate a *value resolver* for us:

```
@TemplateData(target = String.class)
```

Now this works as expected:

```
<p>Name: {name.replace('k', 'c')}</p>
```

Type safe templates

In the previous example we saw that the following line:

```
<p>Name: {name.replace('k', 'c')}</p>
```

printed NOT_FOUND, at run time. We can improve on this, and make Qute generate an error at build time, by indicating in the template that we expect a value of type **String**:

```
1 {@java.lang.String name}
2 <html>
3   <head>
4     <title>Qute Examples</title>
5   </head>
6   <body>
7     <p>Name: {name.replace('k', 'c')}</p>
8   </body>
9 </html>
```

Type safe templates

In the previous example we saw that the following line:

```
<p>Name: {name.replace('k', 'c')}</p>
```

printed NOT_FOUND, at run time. We can improve on this, and make Qute generate an error at build time, by indicating in the template that we expect a value of type **String**:

```
1 {@java.lang.String name}
2 <html>
3   <head>
4     <title>Qute Examples</title>
5   </head>
6   <body>
7     <p>Name: {name.replace('k', 'c')}</p>
8   </body>
9 </html>
```

Type safe templates

In the previous example we saw that the following line:

```
<p>Name: {name.replace('k', 'c')}</p>
```

printed NOT_FOUND, at run time. We can improve on this, and make Qute generate an error at build time, by indicating in the template that we expect a value of type **String**:

```
1 {@java.lang.String name}
2 <html>
3   <head>
4     <title>Qute Examples</title>
5   </head>
6   <body>
7     <p>Name: {name.replace('k', 'c')}</p>
8   </body>
9 </html>
```

Type-safe templates

Now, Qute will render an error:

Error restarting Quarkus

Found 1 Qute problems

#1 Incorrect expression found: {name.replace('k','c')}

- property/method [replace('k','c')] not found on class [java.lang.String] nor handled by an extension method
- at quoteExamples.html:7

```
1  {@java.lang.String name}
2  <html>
3  <head>
4      <title>Qute Examples</title>
5  </head>
6  <body>
7  <p>Name: {name.replace('k', 'c')}</p>
8  =====^
9  </body>
</html>
```

Exercise: Quite products, part 1

Exercise: Quite products, part 2

Recap

In this module we have:

- Discussed why Qute was created and how it compares to other template engines
- Created a Qute template and used it from a Resource
- Seen how to create custom tags and extension methods
- Seen how to create type-safe templates

Persistence

Learning outcomes

After this module, you should:

- Understand the different connectivity options
- Know how to configure a data source
- Know how to use Hibernate + Panache to retrieve and store data
- Understand where `@Transactional` annotation can be placed

Multiple options for persistence layer

- Hibernate ORM and JPA
- Hibernate ORM with Panache
 - will use for "imperative" part of the course
- Reactive SQL
 - will use for "reactive" part of the course
- Many NoSQL clients (MongoDB, Redis, Neo4j, Cassandra, etc.)

Quarkus, Hibernate & Panache

- Quarkus and Hibernate are best buddies.
- Panache is a layer on top which:
 - Facilitates Active Record or Repository patterns
 - ... with many pre-created methods
 - Can create RESTful endpoints for entities

Configuring the Data Source

- Agroal is the default datasource connection pooling implementation for configuring with JDBC driver
- `quarkus.datasource.*` keys in `application.properties`

```
quarkus.datasource.db-kind=...
quarkus.datasource.username=...
quarkus.datasource.password=...
quarkus.datasource.jdbc.url=...
```

Active Record Example

```
1 @Entity
2 public class Product extends PanacheEntity {
3
4     public String name;
5     public String description;
6     public BigDecimal price;
7
8     public static Product findByName(String name){
9         return find("name", name).firstResult();
10    }
11
12    public static List<Product> findExpensive(){
13        return list("price > ?1", new BigDecimal("100"));
14    }
15
16    public static void deleteChairs(){}
```

Active Record Example

```
1 @Entity
2 public class Product extends PanacheEntity {
3
4     public String name;
5     public String description;
6     public BigDecimal price;
7
8     public static Product findByName(String name){
9         return find("name", name).firstResult();
10    }
11
12    public static List<Product> findExpensive(){
13        return list("price > ?1", new BigDecimal("100"));
14    }
15
16    public static void deleteChairs(){}
```

Active Record Example

```
1  package;
2  public class Product extends PanacheEntity {
3
4      public String name;
5      public String description;
6      public BigDecimal price;
7
8      public static Product findByName(String name){
9          return find("name", name).firstResult();
10     }
11
12     public static List<Product> findExpensive(){
13         return list("price > ?1", new BigDecimal("100"));
14     }
15
16     public static void deleteChairs(){
17         delete("name = "Chair");
```

Active Record Example

```
4     public String name;
5     public String description;
6     public BigDecimal price;
7
8     public static Product findByName(String name){
9         return find("name", name).firstResult();
10    }
11
12    public static List<Product> findExpensive(){
13        return list("price > ?1", new BigDecimal("100"));
14    }
15
16    public static void deleteChairs(){
17        delete("name", "Chair");
18    }
19 }
```

Active Record Example

```
4     public String name;
5     public String description;
6     public BigDecimal price;
7
8     public static Product findByName(String name){
9         return find("name", name).firstResult();
10    }
11
12    public static List<Product> findExpensive(){
13        return list("price > ?1", new BigDecimal("100"));
14    }
15
16    public static void deleteChairs(){
17        delete("name", "Chair");
18    }
19 }
```

Active Record Example

```
1 @Entity
2 public class Product extends PanacheEntity {
3
4     public String name;
5     public String description;
6     public BigDecimal price;
7
8     public static Product findByName(String name){
9         return find("name", name).firstResult();
10    }
11
12    public static List<Product> findExpensive(){
13        return list("price > ?1", new BigDecimal("100"));
14    }
15
16    public static void deleteChairs(){}
```

Repository Example

```
@ApplicationScoped
public class ProductRepository implements PanacheRepository<Product> {

    public Product findByName(String name){
        return find("name", name).firstResult();
    }

    public List<Person> findExpensive(){
        return list("price > ?1", new BigDecimal("100"));
    }

    public void deleteChairs(){
        delete("name", "Chair");
    }
}
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Paging

```
1 PanacheQuery<Product> activeProducts = Product.find("status", Status.Active);
2
3 activeProducts.page(Page.ofSize(25));
4
5 List<Product> firstPage = activeProducts.list();
6
7 List<Product> secondPage = activeProducts.nextPage().list();
8
9 List<Product> page7 = activeProducts.page(Page.of(7, 25)).list();
10
11 int numberOfPages = activeProducts.pageCount();
12
13 long count = activeProducts.count();
14
15 return Product.find("status", Status.Alive)
16 .page(Page.ofSize(25))
```

Sorting

```
public static List<Product> findExpensive() {
    return list(Sort.by("price"));
}
```

```
public static List<Product> findExpensive() {
    return list("price > ?1", Sort.by("price").descending(), new BigDecimal("100"));
}
```

Query Projection

```
@RegisterForReflection
public class ProductName {
    public final String name;

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

```
PanacheQuery<ProductName> query = Product.find("active", Status.Active).project(Product
```

Field access rewrite

- You can write your Panache Entity with public fields
- Quarkus will automatically rewrite all access to (generated) getters and setters
- You can override getters and setters when you want.

Field access rewrite example

```
1 public class Product extends PanacheEntity {  
2  
3     public String name;  
4     public String description;  
5     public BigDecimal price;  
6  
7     public String getName() {  
8         return name.toUpperCase();  
9     }  
10  
11 }
```

Elsewhere:

```
System.out.println(product.name);
```

Field access rewrite example

```
1 public class Product extends PanacheEntity {  
2  
3     public String name;  
4     public String description;  
5     public BigDecimal price;  
6  
7     public String getName() {  
8         return name.toUpperCase();  
9     }  
10  
11 }
```

Elsewhere:

```
System.out.println(product.name);
```

Field access rewrite example

```
1 public class Product extends PanacheEntity {  
2  
3     public String name;  
4     public String description;  
5     public BigDecimal price;  
6  
7     public String getName() {  
8         return name.toUpperCase();  
9     }  
10  
11 }
```

Elsewhere:

```
System.out.println(product.name);
```

Transactions

- All persistence-related extensions integrate the Transaction Manager
- Declarative approach with `@Transactional` annotation
- Six transactional types
 - REQUIRED (which is the default)
 - REQUIRED_NEW
 - MANDATORY
 - SUPPORTS
 - NOT_SUPPORTED
 - NEVER

Putting our products in the database

For development:

- `import.sql` in the `src/main/resources` folder
- `quarkus.hibernate-orm.database.generation=drop-and-create`

In production we would use schema migration with Flyway

Exercise: Product from the
database

CDI & ArC

CDI is a Jakarta EE and MicroProfile spec for Context & Dependency Injection

Quarkus has a partial implementation called *ArC*

ArC - Build Time DI

- At compile-time ArC analyzes all classes and dependencies
- At runtime, ArC just has to read the generated metadata and instantiate classes.

Features

- Field, constructor and setter injection
- @Dependent, @ApplicationScoped, @Singleton, @RequestScoped and @SessionScoped scopes
- @AroundInvoke, @PostConstruct, @PreDestroy, @AroundConstruct lifecycle callbacks and interceptors

Injection

- Basically what you'd expect
- Uses `@Inject` annotation
- All resolved compile time, so no general `@Conditional` like Spring
- But there is `@IfBuildProperty`

Scopes

- Normal scopes: `@ApplicationScoped`, `@RequestScoped`,
`@SessionScoped` - Created when a method is invoked.
- Pseudo scopes: `@Singleton`, `@Dependent` - Created when injected.

Lifecycle callbacks

```
@ApplicationScoped
public class MyBean {

    @PostConstruct
    void init() {
        System.out.println("MyBean created!");
    }
}
```

Interceptors

Create an *Interceptor Binding*

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Timed {}
```

and an Interceptor:

```
1 @Timed
2 @Priority(100)
3 @Interceptor
4 public class TimedInterceptor {
5
6     @AroundInvoke
7     Object timeInvocation(InvocationContext context) {
8         long start = System.currentTimeMillis();
9         try {
10             Object ret = context.proceed();
11             long duration = System.currentTimeMillis() - start;
12             System.out.println(context.getMethod().getName() + " call took " + durat
13             return ret;
14         } catch(Exception e) {
15             throw new RuntimeException(e);
16         }
```

Interceptors

Create an *Interceptor Binding*

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Timed {}
```

and an Interceptor:

```
1 @Timed
2 @Priority(100)
3 @Interceptor
4 public class TimedInterceptor {
5
6     @AroundInvoke
7     Object timeInvocation(InvocationContext context) {
8         long start = System.currentTimeMillis();
9         try {
10             Object ret = context.proceed();
11             long duration = System.currentTimeMillis() - start;
12             System.out.println(context.getMethod().getName() + " call took " + duration);
13             return ret;
14         } catch(Exception e) {
15             throw new RuntimeException(e);
16         }
17     }
18 }
```

Interceptors

Create an *Interceptor Binding*

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Timed {}
```

and an Interceptor:

```
1 @Timed
2 @Priority(100)
3 @Interceptor
4 public class TimedInterceptor {
5
6     @AroundInvoke
7     Object timeInvocation(InvocationContext context) {
8         long start = System.currentTimeMillis();
9         try {
10             Object ret = context.proceed();
11             long duration = System.currentTimeMillis() - start;
12             System.out.println(context.getMethod().getName() + " call took " + duration);
13             return ret;
14         } catch(Exception e) {
15             throw new RuntimeException(e);
16         }
17     }
18 }
```

Interceptors

Create an *Interceptor Binding*

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Timed {}
```

and an Interceptor:

```
4 public class TimedInterceptor {
5
6     @AroundInvoke
7     Object timeInvocation(InvocationContext context) {
8         long start = System.currentTimeMillis();
9         try {
10             Object ret = context.proceed();
11             long duration = System.currentTimeMillis() - start;
12             System.out.println(context.getMethod().getName() + " call took " + duration);
13             return ret;
14         } catch(Exception e) {
15             throw new RuntimeException(e);
16         }
17     }
18 }
19 }
```

Interceptors

Create an *Interceptor Binding*

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Timed {}
```

and an Interceptor:

```
1 @Timed
2 @Priority(100)
3 @Interceptor
4 public class TimedInterceptor {
5
6     @AroundInvoke
7     Object timeInvocation(InvocationContext context) {
8         long start = System.currentTimeMillis();
9         try {
10             Object ret = context.proceed();
11             long duration = System.currentTimeMillis() - start;
12             System.out.println(context.getMethod().getName() + " call took " + durat
13             return ret;
14         } catch(Exception e) {
15             throw new RuntimeException(e);
16         }
```

Interceptors

Now we can use it on a method:

```
@Timed
public static List<Product> getAll() {
    return listAll();
}
```

Alternatively, we could put it on the class, to time all method invocations on the class.

Non-standard Features

- `@Inject` can be skipped if there's a qualifier annotation like `@ConfigProperty` present
- No-args constructors can be skipped, and `@Inject` is not needed if there's only one constructor
- Mark a bean intended to be overridden as `@DefaultBean`

Exercise: CDI & ArC

Recap

In this module we have:

- Configured a Quarkus datasource to connect to PostgreSQL
- Added the Hibernate+Panache extension for our persistence layer
- Seen how to use the `@Transactional` annotation
- Explored Dependency Injection with CDI and ArC

Web Services

Learning outcomes

After this module, you should:

- Know how to write a GET endpoint that returns JSON
- Know how to write a POST endpoint that takes and validates JSON
- Know how to generate an Open API spec
- Know how to use `@QuarkusTest` to write an integration test for RESTful endpoint

Introduction

In this chapter we will be updating our Qute endpoints to JSON endpoints, and interact with them using a React frontend.

Automatic JSON serialization

Given a class Greet:

```
public class Greet {  
  
    public final String subject;  
    public final String greet;  
  
    public Greet(String subject, String greet) {  
        this.subject = subject;  
        this.greet = greet;  
    }  
}
```

Automatic JSON serialization

Quarkus and RESTeasy can automatically serialize it to JSON, if you tell it to produce JSON:

```
@Path("hello-json")
public class HelloJsonResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Greet hello() {
        return new Greet("world", "Hello");
    }

}
```

This returns:

```
{
    "subject": "world",
    "greet": "Hello"
}
```

Automatic JSON serialization with Jackson

We can use Jackson annotations to change how the JSON is generated:

```
1 import com.fasterxml.jackson.annotation.JsonProperty;
2
3 public class Greet {
4
5     public final String subject;
6
7     @JsonProperty("TheGreeting")
8     public final String greet;
9
10    public Greet(String subject, String greet) {
11        this.subject = subject;
12        this.greet = greet;
13    }
14
15 }
```

Now we get the following output:

```
{
  "subject": "world",
  "TheGreeting": "Hello"
}
```

Automatic JSON serialization with Jackson

We can use Jackson annotations to change how the JSON is generated:

```
1 import com.fasterxml.jackson.annotation.JsonProperty;
2
3 public class Greet {
4
5     public final String subject;
6
7     @JsonProperty("TheGreeting")
8     public final String greet;
9
10    public Greet(String subject, String greet) {
11        this.subject = subject;
12        this.greet = greet;
13    }
14}
15 }
```

Now we get the following output:

```
{
  "subject": "world",
  "TheGreeting": "Hello"
}
```

Alternatives

But Quarkus isn't tied to Jackson! This just happens to use Jackson, because we have the extension `quarkus-resteasy-jackson` installed. But if we prefer JSON-B instead (part of Microprofile!), we can use the `quarkus-resteasy-jsonb` extension. And then we can use JSON-B annotations:

```
import javax.json.bind.annotation.JsonbProperty;

public class Greet {

    @JsonbProperty("sayWho")
    public final String subject;

    public final String greet;

    public Greet(String subject, String greet) {
        this.subject = subject;
        this.greet = greet;
    }

}
```

Alternatives

We can also have both extensions installed. Quarkus will pick the right serialization framework based on the annotations that are used by the class.

Jackson JSON object

To return JSON, we can use the Jackson library:

```
@Inject  
ObjectMapper mapper;  
  
@GET  
public ObjectNode node() {  
    ObjectNode node = mapper.createObjectNode();  
    node.put("greeting", "Hello");  
    node.put("subject", "Quarkus Students");  
    return node;  
}
```

Exercise: Convert endpoints to
JSON

OpenAPI and Swagger UI

- Simply add SmallRye OpenAPI extension
 - This is an implementation of the MicroProfile Open API spec
- We automatically get /openapi and /swagger-ui
- Can enrich the OpenAPI descriptions with more annotations:
 - @Operation, @APIResponse, @Parameter, @RequestBody, @OpenAPIDefinition
- Swagger UI is good for testing API

Exercise: Add Open API

Exercise: Adding REST data Panache

Exercise: Test your endpoints

Exercise: Hook up the react app

Validation

- Bean Validation can be used to enforce certain constraints
- We can use Hibernate Validator, especially to validate input to REST endpoint we want to add for creating products
 - Simplest way is with an `@Valid` annotation on the request body parameter
- Many standard constraints available under
`javax.validation.constraints.*`

```
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Min;
import javax.validation.constraints.Max;

public class Person {

    @NotBlank
    @NotNull
    public String name;

    @Min(value=0)
    @Max(value=150)
    public int age;
}
```

Exercise: Create PUT endpoint,
enable feature flag and try in the
react app

Recap

In this module we have:

- Seen how JSON serialisation works in Quarkus
- Added a POST endpoint with validation of JSON body
- Generated an OpenAPI spec and seen how to access Swagger UI
- Hooked up a React frontend to our HIQUEA application
- Seen how to use Bean Validation for REST endpoint validation

Reactive Programming

Execution Model

When using the standard *imperative* RESTEasy, Quarkus creates as many **executor threads** as needed, up to the configured maximum:

```
1 @GET
2 @Path("/slow")
3 public String slow() throws InterruptedException {
4     String thread = Thread.currentThread().getName();
5     System.out.println("Thread: " + thread);
6     Thread.sleep(1000);
7     return thread;
8 }
```

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4          min   mean[+/-sd] median    max
5 Connect:      0     0    0.7      0      3
6 Processing: 1002  1009    6.8    1007    1037
7 Waiting:    1002  1009    6.8    1007    1037
8 Total:      1002  1010    7.4    1007    1039
```

The default maximum is `max(200, 8 * nr_of_cores)`

Execution Model

When using the standard *imperative* RESTEasy, Quarkus creates as many executor threads as needed, up to the configured maximum:

```
1 @GET
2 @Path("/slow")
3 public String slow() throws InterruptedException {
4     String thread = Thread.currentThread().getName();
5     System.out.println("Thread: " + thread);
6     Thread.sleep(1000);
7     return thread;
8 }
```

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4          min   mean[+/-sd] median   max
5 Connect:      0     0    0.7      0     3
6 Processing: 1002  1009    6.8   1007   1037
7 Waiting:    1002  1009    6.8   1007   1037
8 Total:      1002  1010    7.4   1007   1039
```

The default maximum is `max(200, 8 * nr_of_cores)`

Execution Model

When using the standard *imperative* RESTEasy, Quarkus creates as many **executor threads** as needed, up to the configured maximum:

```
1 @GET
2 @Path("/slow")
3 public String slow() throws InterruptedException {
4     String thread = Thread.currentThread().getName();
5     System.out.println("Thread: " + thread);
6     Thread.sleep(1000);
7     return thread;
8 }
```

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4          min   mean[+/-sd] median    max
5 Connect:      0     0    0.7      0      3
6 Processing: 1002  1009    6.8   1007   1037
7 Waiting:    1002  1009    6.8   1007   1037
8 Total:      1002  1010    7.4   1007   1039
```

The default maximum is `max(200, 8 * nr_of_cores)`

Execution Model

When using the standard *imperative* RESTEasy, Quarkus creates as many **executor threads** as needed, up to the configured maximum:

```
1 @GET
2 @Path("/slow")
3 public String slow() throws InterruptedException {
4     String thread = Thread.currentThread().getName();
5     System.out.println("Thread: " + thread);
6     Thread.sleep(1000);
7     return thread;
8 }
```

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4             min   mean[+/-sd] median   max
5 Connect:      0     0    0.7      0     3
6 Processing:  1002  1009    6.8    1007   1037
7 Waiting:     1002  1009    6.8    1007   1037
8 Total:       1002  1010    7.4    1007   1039
```

The default maximum is `max(200, 8 * nr_of_cores)`

Execution Model

When using the standard *imperative* RESTEasy, Quarkus creates as many **executor threads** as needed, up to the configured maximum:

```
1 @GET
2 @Path("/slow")
3 public String slow() throws InterruptedException {
4     String thread = Thread.currentThread().getName();
5     System.out.println("Thread: " + thread);
6     Thread.sleep(1000);
7     return thread;
8 }
```

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4          min   mean[+/-sd] median   max
5 Connect:      0     0    0.7      0     3
6 Processing: 1002 1009    6.8    1007   1037
7 Waiting:    1002 1009    6.8    1007   1037
8 Total:      1002 1010    7.4    1007   1039
```

The default maximum is `max(200, 8 * nr_of_cores)`

Execution Model

When using the standard *imperative* RESTEasy, Quarkus creates as many **executor threads** as needed, up to the configured maximum:

```
1 @GET
2 @Path("/slow")
3 public String slow() throws InterruptedException {
4     String thread = Thread.currentThread().getName();
5     System.out.println("Thread: " + thread);
6     Thread.sleep(1000);
7     return thread;
8 }
```

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4          min   mean[+/-sd] median    max
5 Connect:      0     0    0.7      0      3
6 Processing: 1002  1009    6.8   1007   1037
7 Waiting:    1002  1009    6.8   1007   1037
8 Total:      1002  1010    7.4   1007   1039
```

The default maximum is `max(200, 8 * nr_of_cores)`

Execution Model

If we choose a smaller amount of maximum threads:

`quarkus.thread-pool.max-threads=10`

Then running the same ab command takes much longer:

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4      min  mean[+/-sd] median   max
5 Connect:        0    0    0.5     0      2
6 Processing:  1020  4679  959.9   5021   5068
7 Waiting:     1020  4679  960.0   5020   5068
8 Total:       1022  4680  959.5   5021   5070
```

Execution Model

If we choose a smaller amount of maximum threads:

`quarkus.thread-pool.max-threads=10`

Then running the same ab command takes much longer:

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4      min  mean[+/-sd] median   max
5 Connect:        0    0     0.5     0      2
6 Processing:  1020  4679  959.9   5021   5068
7 Waiting:     1020  4679  960.0   5020   5068
8 Total:       1022  4680  959.5   5021   5070
```

Execution Model

If we choose a smaller amount of maximum threads:

`quarkus.thread-pool.max-threads=10`

Then running the same ab command takes much longer:

```
1 ab -c 50 -n300 http://127.0.0.1:8081/threads/slow
2 ...
3 Connection Times (ms)
4      min  mean[+/-sd] median   max
5 Connect:        0    0    0.5     0      2
6 Processing:  1020  4679  959.9   5021   5068
7 Waiting:     1020  4679  960.0   5020   5068
8 Total:       1022  4680  959.5   5021   5070
```

Execution Model - Blocking Threads

Two types of a thread being held up:

- Doing useful work on the CPU
- Waiting for somebody else (Database, API call, Disk IO, etc.). This is what we call *blocking*.

Execution Model - Blocking Threads

Two types of a thread being held up:

- Doing useful work on the CPU
- Waiting for somebody else (Database, API call, Disk IO, etc.). This is what we call *blocking*.

RESTEasy Reactive

The `quarkus-resteasy-reactive` extension brings reactive JAX-RS support to Quarkus.

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

```
@GET
@Path("/hello")
public String hello() {
    return "Hello World";
}
```

Works identically.

RESTEasy Reactive

The `quarkus-resteasy-reactive` extension brings reactive JAX-RS support to Quarkus.

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

```
1 @GET
2 @Path("/hello")
3 @Produces(MediaType.TEXT_PLAIN)
4 public CompletionStage<String> hello() {
5     return CompletableFuture.completedFuture("Hello!");
6 }
```

We can also return a `CompletionStage`

RESTEasy Reactive

The `quarkus-resteasy-reactive` extension brings reactive JAX-RS support to Quarkus.

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

```
1 @GET
2 @Path("/hello")
3 @Produces(MediaType.TEXT_PLAIN)
4 public CompletionStage<String> hello() {
5     return CompletableFuture.completedFuture("Hello!");
6 }
```

We can also return a `CompletionStage`

RESTEasy Reactive

The `quarkus-resteasy-reactive` extension brings reactive JAX-RS support to Quarkus.

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

```
1 @GET
2 @Path("/hello")
3 @Produces(MediaType.TEXT_PLAIN)
4 public CompletionStage<String> hello() {
5     return CompletableFuture.completedFuture("Hello!");
6 }
```

We can also return a `CompletionStage`

Reactive Execution Model

RESTEasy reactive **does** care about blocking

- Your method will be called by a Vert.x eventloop thread
- You shouldn't block it
- If you do block, annotate with `@Blocking`

Reactive Execution Model

Example

```
@GET  
@Path( "/regular" )  
public String regular() {  
    return Thread.currentThread().getName();  
}
```

This is fine - returns something like `vert.x-eventloop-thread-3`

Reactive Execution Model

Bad Example

```
1 @GET
2 @Path("/regular-slow")
3 public String regularSlow() {
4     Thread.sleep(1000);
5     return Thread.currentThread().getName();
6 }
```

This is **not** fine.

Reactive Execution Model

Bad Example

```
1 @GET
2 @Path("/regular-slow")
3 public String regularSlow() {
4     Thread.sleep(1000);
5     return Thread.currentThread().getName();
6 }
```

This is **not** fine.

Reactive Execution Model

Bad Example

```
1 @GET
2 @Path("/regular-slow")
3 public String regularSlow() {
4     Thread.sleep(1000);
5     return Thread.currentThread().getName();
6 }
```

This is **not** fine.

Reactive Execution Model

Bad Example

```
1 ab -c50 -n300 http://127.0.0.1:8082/threads/regular-slow
2
3 Connection Times (ms)
4 min mean[+/-sd] median max
5 Connect:      0    1   0.7      0      3
6 Processing:  1005 2013 402.7  2012    3087
7 Waiting:     1005 2012 402.7  2012    3087
8 Total:       1007 2013 402.3  2013    3088
9 WARNING: The median and mean for the initial connection time are not within a normal
10 These results are probably not that reliable.
```

Reactive Execution Model

Bad Example

```
1 ab -c50 -n300 http://127.0.0.1:8082/threads/regular-slow
2
3 Connection Times (ms)
4 min mean[+/-sd] median max
5 Connect:      0    1   0.7      0      3
6 Processing:  1005 2013 402.7  2012    3087
7 Waiting:     1005 2012 402.7  2012    3087
8 Total:       1007 2013 402.3  2013    3088
9 WARNING: The median and mean for the initial connection time are not within a normal
10 These results are probably not that reliable.
```

Reactive Execution Model

Bad Example

```
1 ab -c50 -n300 http://127.0.0.1:8082/threads/regular-slow
2
3 Connection Times (ms)
4 min mean[+/-sd] median max
5 Connect:      0    1   0.7      0      3
6 Processing:  1005 2013 402.7  2012    3087
7 Waiting:     1005 2012 402.7  2012    3087
8 Total:       1007 2013 402.3  2013    3088
9 WARNING: The median and mean for the initial connection time are not within a normal
10 These results are probably not that reliable.
```

Reactive Execution Model

Bad Example

```
1 ab -c50 -n300 http://127.0.0.1:8082/threads/regular-slow
2
3 Connection Times (ms)
4 min mean[+/-sd] median max
5 Connect:      0    1   0.7      0      3
6 Processing:  1005 2013 402.7  2012    3087
7 Waiting:     1005 2012 402.7  2012    3087
8 Total:       1007 2013 402.3  2013    3088
9 WARNING: The median and mean for the initial connection time are not within a normal
10 These results are probably not that reliable.
```

Reactive Execution Model

Bad Example

```
1 ab -c50 -n300 http://127.0.0.1:8082/threads/regular-slow
2
3 Connection Times (ms)
4 min mean[+/-sd] median max
5 Connect:      0    1   0.7      0      3
6 Processing:  1005 2013 402.7  2012    3087
7 Waiting:     1005 2012 402.7  2012    3087
8 Total:       1007 2013 402.3  2013    3088
9 WARNING: The median and mean for the initial connection time are not within a normal
10 These results are probably not that reliable.
```

Reactive Execution Model

Good Example

```
1 @GET
2 @Path("/regular-slow")
3 @Blocking
4 public String blockingSlow() {
5     Thread.sleep(1000);
6     return Thread.currentThread().getName();
7 }
```

This returns **executor-thread-221**

Reactive Execution Model

Good Example

```
1 @GET
2 @Path("/regular-slow")
3 @Blocking
4 public String blockingSlow() {
5     Thread.sleep(1000);
6     return Thread.currentThread().getName();
7 }
```

This returns **executor-thread-221**

Reactive Execution Model

Good Example

```
1 @GET
2 @Path("/regular-slow")
3 @Blocking
4 public String blockingSlow() {
5     Thread.sleep(1000);
6     return Thread.currentThread().getName();
7 }
```

This returns **executor-thread-221**

Reactive Execution Model

Good Example

```
ab -c70 -n300 http://127.0.0.1:8082/threads/blocking-slow

Connection Times (ms)
    min  mean[+/-sd] median   max
Connect:      0     1    0.7     1     3
Processing: 1001 1008    4.5   1007   1023
Waiting:    1001 1008    4.5   1007   1023
Total:       1001 1009    4.9   1008   1024
```

Back to normal :)

Reactive Execution Model

Even better example

Of course, we can do even much better, by not blocking a thread at all:

```
1 @GET
2 @Path("/nonblocking-slow")
3 public Uni<String> nonblockingSlow() {
4     return Uni.createFrom().item(Thread.currentThread().getName())
5         .onItem().delayIt().by(Duration.ofSeconds(1))
6         .onItem().transform(i ->
7             "Initial: " + i + ", later: " + Thread.currentThread().getName());
8 }
```

Outputs: Initial: vert.x-eventloop-thread-18, later: executor-thread-1

Reactive Execution Model

Even better example

Of course, we can do even much better, by not blocking a thread at all:

```
1 @GET
2 @Path( "/nonblocking-slow" )
3 public Uni<String> nonblockingSlow() {
4     return Uni.createFrom().item(Thread.currentThread().getName())
5         .onItem().delayIt().by(Duration.ofSeconds(1))
6         .onItem().transform(i ->
7             "Initial: " + i + ", later: " + Thread.currentThread().getName());
8 }
```

Outputs: Initial: vert.x-eventloop-thread-18, later: executor-thread-1

Reactive Execution Model

Even better example

Of course, we can do even much better, by not blocking a thread at all:

```
1 @GET
2 @Path("/nonblocking-slow")
3 public Uni<String> nonblockingSlow() {
4     return Uni.createFrom().item(Thread.currentThread().getName())
5         .onItem().delayIt().by(Duration.ofSeconds(1))
6         .onItem().transform(i ->
7             "Initial: " + i + ", later: " + Thread.currentThread().getName());
8 }
```

Outputs: Initial: vert.x-eventloop-thread-18, later: executor-thread-1

Reactive Execution Model

Even better example

Of course, we can do even much better, by not blocking a thread at all:

```
1 @GET
2 @Path("/nonblocking-slow")
3 public Uni<String> nonblockingSlow() {
4     return Uni.createFrom().item(Thread.currentThread().getName())
5         .onItem().delayIt().by(Duration.ofSeconds(1))
6         .onItem().transform(i ->
7             "Initial: " + i + ", later: " + Thread.currentThread().getName());
8 }
```

Outputs: Initial: vert.x-eventloop-thread-18, later: executor-thread-1

Reactive Execution Model

Even better example

Of course, we can do even much better, by not blocking a thread at all:

```
1 @GET
2 @Path("/nonblocking-slow")
3 public Uni<String> nonblockingSlow() {
4     return Uni.createFrom().item(Thread.currentThread().getName())
5         .onItem().delayIt().by(Duration.ofSeconds(1))
6         .onItem().transform(i ->
7             "Initial: " + i + ", later: " + Thread.currentThread().getName());
8 }
```

Outputs: Initial: vert.x-eventloop-thread-18, later: executor-thread-1

Reactive Routes

An alternative to *RESTEasy Reactive* is to use the *Reactive Routes* extension:

Reactive routes propose an alternative approach to implement HTTP endpoints where you declare and chain routes. This approach became very popular in the JavaScript world, with frameworks like Express.Js or Hapi. Quarkus also offers the possibility to use reactive routes. You can implement REST API with routes only or combine them with JAX-RS resources and servlets.

Reactive Routes

```
1 @Route(methods = HttpMethod.GET)
2 void hello(RoutingContext rc) {
3     rc.response().end("hello");
4 }
5
6 @Route(path = "/hello")
7 Uni<String> hello(RoutingContext context) {
8     return Uni.createFrom().item("Hello world!");
9 }
10
11 @Route(produces = "application/json")
12 Person createPerson(@Body Person person, @Param("id") Optional<String> primaryKey) {
13     person.setId(primaryKey.map(Integer::valueOf).orElse(42));
14     return person;
15 }
```

Reactive Routes

```
1 @Route(methods = HttpMethod.GET)
2 void hello(RoutingContext rc) {
3     rc.response().end("hello");
4 }
5
6 @Route(path = "/hello")
7 Uni<String> hello(RoutingContext context) {
8     return Uni.createFrom().item("Hello world!");
9 }
10
11 @Route(produces = "application/json")
12 Person createPerson(@Body Person person, @Param("id") Optional<String> primaryKey) {
13     person.setId(primaryKey.map(Integer::valueOf).orElse(42));
14     return person;
15 }
```

Reactive Routes

```
1 @Route(methods = HttpMethod.GET)
2 void hello(RoutingContext rc) {
3     rc.response().end("hello");
4 }
5
6 @Route(path = "/hello")
7 Uni<String> hello(RoutingContext context) {
8     return Uni.createFrom().item("Hello world!");
9 }
10
11 @Route(produces = "application/json")
12 Person createPerson(@Body Person person, @Param("id") Optional<String> primaryKey) {
13     person.setId(primaryKey.map(Integer::valueOf).orElse(42));
14     return person;
15 }
```

Reactive Routes

```
1 @Route(methods = HttpMethod.GET)
2 void hello(RoutingContext rc) {
3     rc.response().end("hello");
4 }
5
6 @Route(path = "/hello")
7 Uni<String> hello(RoutingContext context) {
8     return Uni.createFrom().item("Hello world!");
9 }
10
11 @Route(produces = "application/json")
12 Person createPerson(@Body Person person, @Param("id") Optional<String> primaryKey) {
13     person.setId(primaryKey.map(Integer::valueOf).orElse(42));
14     return person;
15 }
```

Reactive Routes

```
1 @Route(methods = HttpMethod.GET)
2 void hello(RoutingContext rc) {
3     rc.response().end("hello");
4 }
5
6 @Route(path = "/hello")
7 Uni<String> hello(RoutingContext context) {
8     return Uni.createFrom().item("Hello world!");
9 }
10
11 @Route(produces = "application/json")
12 Person createPerson(@Body Person person, @Param("id") Optional<String> primaryKey) {
13     person.setId(primaryKey.map(Integer::valueOf).orElse(42));
14     return person;
15 }
```

Reactive Database Access

About JDBC

JDBC is a blocking API

Example:

```
ResultSet rs = stmt.executeQuery(query);
```

There is **no way** to obtain the **ResultSet** without blocking a thread.

Hibernate going Reactive

In December 2020, Hibernate Reactive was launched:

```
Uni<Book> bookUni = session.find(Book.class, book.id);
bookUni.invoke( book -> System.out.println(book.title + " is a great book!") )
```

It's a reactive API for Hibernate ORM.

Hibernate going Reactive

- Works with non-blocking database clients. Currently the Vert.x clients for Postgres, MySQL and DB2
- Well-integrated with Quarkus
- No implicit blocking lazy loading, but explicit asynchronous operations for fetching associations

Hibernate Reactive + Panache

- Methods that returned `T` or `List<T>` now return `Uni<T>` and `Uni<List<T>>`
- New methods `streamxxx` that return a `Multi<T>`
- Classes live in a new package under
`io.quarkus.hibernate.reactive`

Hibernate Reactive + Panache usage

```
@GET
public Multi<Product> products() {
    return Product.streamAll();
}

@GET
@Path("{productId}")
public Uni<Product> details(@PathParam("productId") Long productId) {
    return Product.findById(productId);
}
```

Mutiny, Uni & Multi

Mutiny is the library for Reactive Programming that Quarkus uses. It's two main types are:

- `Multi<T>` represents a stream of items of type T
- `Uni<T>`, represents a stream of zero or one element of type T

RESTEasy Reactive with Mutiny Uni

Unis are supported as a result type:

```
1 @GET
2 @Produces(MediaType.TEXT_PLAIN)
3 public Uni<String> helloUni() {
4     return Uni.createFrom().item("Hello!");
5 }
```

RESTEasy Reactive with Mutiny Uni

Unis are supported as a result type:

```
1 @GET
2 @Produces(MediaType.TEXT_PLAIN)
3 public Uni<String> helloUni() {
4     return Uni.createFrom().item("Hello!");
5 }
```

RESTEasy Reactive with Mutiny Uni

Unis are supported as a result type:

```
1 @GET
2 @Produces(MediaType.TEXT_PLAIN)
3 public Uni<String> helloUni() {
4     return Uni.createFrom().item("Hello!");
5 }
```

RESTEasy Reactive

Also, Multi is supported:

```
@GET  
@Produces(MediaType.TEXT_PLAIN)  
public Multi<String> helloMulti() {  
    return Multi.createFrom().items("Hello", "world!");  
}
```

This returns a chunked HTTP response.

Exercise: Going Reactive

Sessions & Transactions

```
session.find(Product.class, id)
    .call(product -> session.remove(product))
    .call(() -> session.flush())
```

Sessions & Transactions - Example

Good:

```
Uni<Product> product = session.find(Product.class, id)
    .call(session::remove)
    .call(session::flush)
```

Bad:

```
Uni<Product> product = session.find(Product.class, id)
    .call(session::remove)
    .invoke(session::flush)
```

Methods:

```
Uni<T> call(Supplier<Uni<?>> supplier)
Uni<T> invoke(Runnable callback)
```

Both examples compile and have the right types, but the second one *will never execute the flush.*

Low-level Reactive SQL Clients

Another way of connecting to the DB is using the low-level reactive SQL clients.

```
PgConnectOptions connectOptions = new PgConnectOptions()
    .setPort(5432)
    .setHost("the-host")
    .setDatabase("the-db")
    .setUser("user")
    .setPassword("secret");

// Pool options
PoolOptions poolOptions = new PoolOptions()
    .setMaxSize(5);

// Create the client pool
PgPool client = PgPool.pool(connectOptions, poolOptions);
```

The base object we need is a `PgPool` instance.

Low-level Reactive SQL Clients

```
@Inject
```

```
PgPool client;
```

Pick the right PgPool:

- `io.vertx.mutiny.pgclient.PgPool` uses Mutiny types
- `io.vertx.pgclient.PgPool` uses Vert.x types

Querying

Querying returns a **Uni** containing a **RowSet**:

```
Uni<RowSet<Row>> rowSetUni = client.query("SELECT name, age FROM people").execute();
```

Of course, we can transform this into a **Multi** of **Rows**:

```
Multi<Row> people = client.query("SELECT name, age FROM people").execute()
    .onItem().transformToMulti(set -> Multi.createFrom().iterable(set));
```

Querying

```
Multi<Person> people = client.query("SELECT name, age FROM people")
    .execute()
    .onItem().transformToMulti(rows -> Multi.createFrom().iterable(rows))
    .onItem().transform(Person::fromRow);
```

```
static Person fromRow(Row row) {
    return new Person(row.getString("nam"), row.getInteger("age"));
}
```

Parameters

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

Parameters

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

Parameters

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

Parameters

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

Parameters

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

Inserts and Updates

```
1 Uni<Long> personId = client
2   .preparedQuery("INSERT INTO people (name, age) VALUES ($1, $2) RETURNING id")
3   .execute(Tuple.of(name, age))
4   .onItem().transform(pgRowSet -> pgRowSet.iterator().next().getLong("id"));
5 }
```

Inserts and Updates

```
1 Uni<Long> personId = client
2   .preparedQuery("INSERT INTO people (name, age) VALUES ($1, $2) RETURNING id")
3   .execute(Tuple.of(name, age))
4   .onItem().transform(pgRowSet -> pgRowSet.iterator().next().getLong("id"));
5 }
```

Inserts and Updates

```
1 Uni<Long> personId = client
2   .preparedQuery("INSERT INTO people (name, age) VALUES ($1, $2) RETURNING id")
3   .execute(Tuple.of(name, age))
4   .onItem().transform(pgRowSet -> pgRowSet.iterator().next().getLong("id"));
5 }
```

Inserts and Updates

```
1 Uni<Long> personId = client
2   .preparedQuery("INSERT INTO people (name, age) VALUES ($1, $2) RETURNING id")
3   .execute(Tuple.of(name, age))
4   .onItem().transform(pgRowSet -> pgRowSet.iterator().next().getLong("id"));
5 }
```

Inserts and Updates

```
1 Uni<Long> personId = client
2   .preparedQuery("INSERT INTO people (name, age) VALUES ($1, $2) RETURNING id")
3   .execute(Tuple.of(name, age))
4   .onItem().transform(pgRowSet -> pgRowSet.iterator().next().getLong("id"));
5 }
```

Inserts and Updates

```
1 Uni<Long> personId = client
2   .preparedQuery("INSERT INTO people (name, age) VALUES ($1, $2) RETURNING id")
3   .execute(Tuple.of(name, age))
4   .onItem().transform(pgRowSet -> pgRowSet.iterator().next().getLong("id"));
5 }
```

Exercise: Reactive search endpoint

Listen & Notify

One of the cool features of Postgres is to Listen to channels. As part of transactions you can notify channels, for example to alert consumers that are waiting for event.

Listen & Notify

```
@Path("/listen/{channel}")
@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
@RestSseElementType(MediaType.APPLICATION_JSON)
public Multi<JsonObject> listen(@PathParam("channel") String channel) {
    return client.getConnection()
        .onItem().transformToMulti(connection -> {
            Multi<PgNotification> notifications = Multi.createFrom().
                emitter(c -> toPgConnection(connection).notificationHandler(c::emit));
            return connection.query("LISTEN " + channel).execute().onItem().transformToMulti(
            }).map(PgNotification::toJson);
}
```

```
@Path("/notify/{channel}")
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.WILDCARD)
public Uni<String> notif(@PathParam("channel") String channel, String stuff) {
    return client.preparedQuery("NOTIFY " + channel + ", $$" + stuff + "$$").execute()
        .map(rs -> "Posted to " + channel + " channel");
}
```

Listen & Notify

```
→ http localhost:8082/db/listen/milkshakes --stream
HTTP/1.1 200 OK
Content-Type: text/event-stream
X-SSE-Content-Type: application/json
transfer-encoding: chunked

data:{"channel":"milkshakes","payload":{"\\"flavour\\": \"banana\"}","processId":57}

data:{"channel":"milkshakes","payload":{"\\"flavour\\": \"strawberry\"}","processId":58}
```

```
→ http POST localhost:8082/db/notify/milkshakes flavour=banana
HTTP/1.1 200 OK
Content-Type: text/plain
content-length: 28
```

Posted to milkshakes channel

```
→ http POST localhost:8082/db/notify/milkshakes flavour=strawberry
HTTP/1.1 200 OK
Content-Type: text/plain
content-length: 28
```

Posted to milkshakes channel

Exercise: Listen & Notify

Reactive Streams

Reactive Streams

Streaming data is frequently found in modern applications:

- Events that flow from a system to consumers
- Records that flow from a database into a chunked HTTP response
- Messages that are consumed from a queue, transformed and put on another queue

Reactive Streams

Streaming data is frequently found in modern applications:

- Events that flow from a system to consumers
- Records that flow from a database into a chunked HTTP response
- Messages that are consumed from a queue, transformed and put on another queue

A fundamental problem of streaming data systems, is to make sure that the *consumer* of the stream can handle the messages that are being sent to it.

Slow consumer

Suppose you have a system that reads records from a database,
transforms them to JSON and stores them in a file.

Question: What happens if you read 1000 records per second, but you
can only write 500 per second to files?

Slow consumer - solutions

Possible solutions:

Slow consumer - solutions

Possible solutions:

- Have a really fast consumer instead

Slow consumer - solutions

Possible solutions:

- Have a really fast consumer instead
- Have a really slow producer

Slow consumer - solutions

Possible solutions:

- Have a really fast consumer instead
- Have a really slow producer
- Have more memory than your database size

Slow consumer - solutions

Possible solutions:

- Have a really fast consumer instead
- Have a really slow producer
- Have more memory than your database size
- Adapt the speed of the producer, based on the capacity of the consumer

Back pressure

Back pressure means that the consumer can indicate *demand* to the producer. The producer will only produce the amount that the consumer requested.

Back pressure

- Works for the entire stream, not just the consumer at the end
- Each element can adapt the demand that's sent upstream
 - Slow components reduce demand
 - Some components, like buffers, can increase demand

Streaming across TCP

- TCP natively supports back-pressure!
- *ack* messages contain a *window* field, indicating how much the sender may send.
- When the receiver processed data, a new *ack* gets sent, with a bigger window.

The Reactive Streams standard

Around 2013, engineers from Netflix, Pivotal, Lightbend, Twitter and others were all working on streaming systems, essentially solving the same issues.

To make sure their libraries would be interoperable, they came up with the **Reactive Streams** standard.

It's a minimal interface needed to connect streaming libraries, retaining full non-blocking operation and back-pressure and a shared cancellation and error model.

The Reactive Streams standard

Ended up into the Java Standard Library as of Java 9, under
`java.util.concurrent.Flow`.

Reactive Streams implementations

- Akka Streams
- RxJava
- Reactor
- Vert.x
- Mutiny
- ... and others

Getting Started

```
1 Multi<String> greeter = Multi.createFrom().items("Hello", "world");
2 Uni<List<String>> out = greeter.collectItems().asList();
3 List<String> results = out.subscribe().asCompletionStage().join();
4 System.out.println(results);
```

Getting Started

```
1 Multi<String> greeter = Multi.createFrom().items("Hello", "world");
2 Uni<List<String>> out = greeter.collectItems().asList();
3 List<String> results = out.subscribe().asCompletionStage().join();
4 System.out.println(results);
```

Getting Started

```
1 Multi<String> greeter = Multi.createFrom().items("Hello", "world");
2 Uni<List<String>> out = greeter.collectItems().asList();
3 List<String> results = out.subscribe().asCompletionStage().join();
4 System.out.println(results);
```

Getting Started

```
1 Multi<String> greeter = Multi.createFrom().items("Hello", "world");
2 Uni<List<String>> out = greeter.collectItems().asList();
3 List<String> results = out.subscribe().asCompletionStage().join();
4 System.out.println(results);
```

Mutiny Uni

A Uni only gets executed when connected to a *subscriber*:

```
1 Uni<Integer> myUni = Uni.createFrom().item(() -> {
2     System.out.println("Creating the item!");
3     return 5;
4 });
5
6 // Nothing has been printed at this point
7
8 System.out.println("Subscribing:");
9 myUni.subscribe().with(System.out::println); // Prints 'Creating the item!' and '5'
```

Mutiny Uni

A Uni only gets executed when connected to a *subscriber*:

```
1 Uni<Integer> myUni = Uni.createFrom().item(() -> {
2     System.out.println("Creating the item!");
3     return 5;
4 });
5
6 // Nothing has been printed at this point
7
8 System.out.println("Subscribing:");
9 myUni.subscribe().with(System.out::println); // Prints 'Creating the item!' and '5'
```

Mutiny Multi

Mutiny's `Multi` interface *extends* `org.reactivestreams.Publisher<T>`, so it's a reactive stream.

Multi has many methods to operate on it:

```
1 return Multi.createFrom().items("One", "Two", "Three", "Four", "Five", "Six")
2   .map(String::toUpperCase)
3   .filter(s -> s.length() >= 4)
4   .flatMap(s -> Multi.createFrom().items(s.toCharArray()))
5   .map(String::valueOf);
```

Mutiny Multi

Mutiny's `Multi` interface *extends* `org.reactivestreams.Publisher<T>`, so it's a reactive stream.

Multi has many methods to operate on it:

```
1 return Multi.createFrom().items("One", "Two", "Three", "Four", "Five", "Six")
2   .map(String::toUpperCase)
3   .filter(s -> s.length() >= 4)
4   .flatMap(s -> Multi.createFrom().items(s.toCharArray()))
5   .map(String::valueOf);
```

Mutiny Multi

Mutiny's `Multi` interface *extends* `org.reactivestreams.Publisher<T>`, so it's a reactive stream.

Multi has many methods to operate on it:

```
1 return Multi.createFrom().items("One", "Two", "Three", "Four", "Five", "Six")
2   .map(String::toUpperCase)
3   .filter(s -> s.length() >= 4)
4   .flatMap(s -> Multi.createFrom().items(s.toCharArray()))
5   .map(String::valueOf);
```

Mutiny Multi

Mutiny's `Multi` interface *extends* `org.reactivestreams.Publisher<T>`, so it's a reactive stream.

Multi has many methods to operate on it:

```
1 return Multi.createFrom().items("One", "Two", "Three", "Four", "Five", "Six")
2   .map(String::toUpperCase)
3   .filter(s -> s.length() >= 4)
4   .flatMap(s -> Multi.createFrom().items(s.toCharArray()))
5   .map(String::valueOf);
```

Mutiny Multi

Mutiny's `Multi` interface *extends* `org.reactivestreams.Publisher<T>`, so it's a reactive stream.

Multi has many methods to operate on it:

```
1 return Multi.createFrom().items("One", "Two", "Three", "Four", "Five", "Six")
2   .map(String::toUpperCase)
3   .filter(s -> s.length() >= 4)
4   .flatMap(s -> Multi.createFrom().items(s.toCharArray()))
5   .map(String::valueOf);
```

Mutiny Multi

A subscriber to a Multi, must deal with the following situations:

- An element arrives
- A failure occurred
- The (bounded) stream completed

```
1 Cancellable cancellable = multi
2 .subscribe().with(
3     item -> System.out.println(item),
4     failure -> System.out.println("Failed with " + failure),
5     () -> System.out.println("Completed"));
```

Mutiny Multi

A subscriber to a Multi, must deal with the following situations:

- An element arrives
- A failure occurred
- The (bounded) stream completed

```
1 Cancellable cancellable = multi
2 .subscribe().with(
3     item -> System.out.println(item),
4     failure -> System.out.println("Failed with " + failure),
5     () -> System.out.println("Completed"));
```

Mutiny Multi

A subscriber to a Multi, must deal with the following situations:

- An element arrives
- A failure occurred
- The (bounded) stream completed

```
1 Cancellable cancellable = multi
2 .subscribe().with(
3     item -> System.out.println(item),
4     failure -> System.out.println("Failed with " + failure),
5     () -> System.out.println("Completed"));
```

Mutiny Multi

A subscriber to a Multi, must deal with the following situations:

- An element arrives
- A failure occurred
- The (bounded) stream completed

```
1 Cancellable cancellable = multi
2 .subscribe().with(
3     item -> System.out.println(item),
4     failure -> System.out.println("Failed with " + failure),
5     () -> System.out.println("Completed"));
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Multi.createFrom().items(1,2,3)
2 .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3 .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4 .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5 .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6 .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7 .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8 .subscribe().with(__ -> {}); // Drain
```

This prints the following:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 9223372036854775807
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3)
2     .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3     .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4     .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5     .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6     .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7     .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8     .subscribe().asStream();
9
10 Set<Integer> set = out.collect(Collectors.toSet());
```

This prints:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 256
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3)
2     .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3     .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4     .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5     .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6     .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7     .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8     .subscribe().asStream();
9
10 Set<Integer> set = out.collect(Collectors.toSet());
```

This prints:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 256
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3)
2     .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3     .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4     .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5     .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6     .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7     .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8     .subscribe().asStream();
9
10 Set<Integer> set = out.collect(Collectors.toSet());
```

This prints:

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 256
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬇️ Received item: 3
6 ⬇️ Completed
```

Visualising the events

We can configure the amount of items to queue:

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3)
2   .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3   .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4   .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5   .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6   .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7   .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8   // Use a buffer capacity of 2
9   .subscribe().asStream(2, () -> new ArrayBlockingQueue<>(2));
10
11 Set<Integer> set = out.collect(Collectors.toSet());
```

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 2
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬆️ Requested: 2
6 ⬇️ Received item: 3
7 ⬇️ Completed
```

Visualising the events

We can configure the amount of items to queue:

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3)
2   .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3   .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4   .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5   .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6   .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7   .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8   // Use a buffer capacity of 2
9   .subscribe().asStream(2, () -> new ArrayBlockingQueue<>(2));
10
11 Set<Integer> set = out.collect(Collectors.toSet());
```

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 2
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬆️ Requested: 2
6 ⬇️ Received item: 3
7 ⬇️ Completed
```

Visualising the events

We can configure the amount of items to queue:

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3)
2   .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3   .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4   .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5   .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6   .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7   .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8   // Use a buffer capacity of 2
9   .subscribe().asStream(2, () -> new ArrayBlockingQueue<>(2));
10
11 Set<Integer> set = out.collect(Collectors.toSet());
```

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 2
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬆️ Requested: 2
6 ⬇️ Received item: 3
7 ⬇️ Completed
```

Visualising the events

We can configure the amount of items to queue:

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3)
2   .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3   .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4   .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5   .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6   .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7   .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8   // Use a buffer capacity of 2
9   .subscribe().asStream(2, () -> new ArrayBlockingQueue<>(2));
10
11 Set<Integer> set = out.collect(Collectors.toSet());
```

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 2
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬆️ Requested: 2
6 ⬇️ Received item: 3
7 ⬇️ Completed
```

Visualising the events

The Stream created by `asStream` will *cancel* the Multi when `closed`.

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3,4,5,6)
2   .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3   .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4   .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5   .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6   .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7   .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8   .subscribe().asStream(2, () -> new ArrayBlockingQueue<>(2));
9
10 Set<Integer> set = out.limit(2).collect(Collectors.toSet());
11 out.close();
```

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 2
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬆️ Requested: 2
6 ⬇️ Received item: 3
7 ⬇️ Received item: 4
8 ⬆️ Cancelled
```

Visualising the events

The Stream created by `asStream` will *cancel* the Multi when `closed`.

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3,4,5,6)
2   .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3   .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4   .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5   .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6   .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7   .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8   .subscribe().asStream(2, () -> new ArrayBlockingQueue<>(2));
9
10 Set<Integer> set = out.limit(2).collect(Collectors.toSet());
11 out.close();
```

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 2
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬆️ Requested: 2
6 ⬇️ Received item: 3
7 ⬇️ Received item: 4
8 ⬆️ Cancelled
```

Visualising the events

The Stream created by `asStream` will *cancel* the Multi when `closed`.

```
1 Stream<Integer> out = Multi.createFrom().items(1,2,3,4,5,6)
2   .onSubscribe().invoke(() -> System.out.println("⬇️ Subscribed"))
3   .onItem().invoke(i -> System.out.println("⬇️ Received item: " + i))
4   .onFailure().invoke(f -> System.out.println("⬇️ Failed with " + f))
5   .onCompletion().invoke(() -> System.out.println("⬇️ Completed"))
6   .onCancellation().invoke(() -> System.out.println("⬆️ Cancelled"))
7   .onRequest().invoke(l -> System.out.println("⬆️ Requested: " + l))
8   .subscribe().asStream(2, () -> new ArrayBlockingQueue<>(2));
9
10 Set<Integer> set = out.limit(2).collect(Collectors.toSet());
11 out.close();
```

```
1 ⬇️ Subscribed
2 ⬆️ Requested: 2
3 ⬇️ Received item: 1
4 ⬇️ Received item: 2
5 ⬆️ Requested: 2
6 ⬇️ Received item: 3
7 ⬇️ Received item: 4
8 ⬆️ Cancelled
```

Backpressure strategies

Essentially, there are three things we can do when the producer is faster than the consumer:

- Reduce the speed of the producer
- Buffer items
- Drop items

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Backpressure strategies

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(10))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5
6   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem()
7     .delayIt().by(Duration.ofSeconds(1))).concatenate()
8
9   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
10  .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
11  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
12  .subscribe().with(__ -> {});`
```

Prints the following:

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 1
3 A -  Received item: 0
4 A -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
5 B -  Failed with io.smallrye.mutiny.subscription.BackPressureFailure: Could not emit
```

Dropping excessive elements

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A - ⬇ Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A - ⬇ Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A - ⬆ Requested: " + l))
5   // Drop on overflow
6   .onOverflow().drop()
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Durat
8     .onItem().invoke(i -> System.out.println("B - ⬇ Received item: " + i))
9     .onFailure().invoke(f -> System.out.println("B - ⬇ Failed with " + f))
10    .onRequest().invoke(l -> System.out.println("B - ⬆ Requested: " + l))
11    .subscribe().with(__ -> {});
```

```
1 B - ⬆ Requested: 9223372036854775807
2 A - ⬆ Requested: 9223372036854775807
3 A - ⬇ Received item: 0
4 A - ⬇ Received item: 1
5 B - ⬇ Received item: 0
6 A - ⬇ Received item: 2
7 A - ⬇ Received item: 3
8 B - ⬇ Received item: 2
```

Dropping excessive elements

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5   // Drop on overflow
6   .onOverflow().drop()
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Duration.ofMillis(100)))
8   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
9   .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
10  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
11  .subscribe().with(__ -> {});
```

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 9223372036854775807
3 A -  Received item: 0
4 A -  Received item: 1
5 B -  Received item: 0
6 A -  Received item: 2
7 A -  Received item: 3
8 B -  Received item: 2
```

Dropping excessive elements

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A - ⬇ Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A - ⬇ Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A - ⬆ Requested: " + l))
5   // Drop on overflow
6   .onOverflow().drop()
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Durat
8     .onItem().invoke(i -> System.out.println("B - ⬇ Received item: " + i))
9     .onFailure().invoke(f -> System.out.println("B - ⬇ Failed with " + f))
10    .onRequest().invoke(l -> System.out.println("B - ⬆ Requested: " + l))
11    .subscribe().with(__ -> {});
```

```
1 B - ⬆ Requested: 9223372036854775807
2 A - ⬆ Requested: 9223372036854775807
3 A - ⬇ Received item: 0
4 A - ⬇ Received item: 1
5 B - ⬇ Received item: 0
6 A - ⬇ Received item: 2
7 A - ⬇ Received item: 3
8 B - ⬇ Received item: 2
```

Dropping excessive elements

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5   // Drop on overflow
6   .onOverflow().drop()
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Duration.ofMillis(100)))
8   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
9   .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
10  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
11  .subscribe().with(__ -> {});
```

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 9223372036854775807
3 A -  Received item: 0
4 A -  Received item: 1
5 B -  Received item: 0
6 A -  Received item: 2
7 A -  Received item: 3
8 B -  Received item: 2
```

Dropping excessive elements

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5   // Drop on overflow
6   .onOverflow().drop()
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Duration.ofMillis(900)))
8   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
9   .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
10  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
11  .subscribe().with(__ -> {});
```

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 9223372036854775807
3 A -  Received item: 0
4 A -  Received item: 1
5 B -  Received item: 0
6 A -  Received item: 2
7 A -  Received item: 3
8 B -  Received item: 2
```

Dropping excessive elements

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5   // Drop on overflow
6   .onOverflow().drop()
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Duration.ofMillis(100)))
8   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
9   .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
10  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
11  .subscribe().with(__ -> {});
```

```
1 B -  Requested: 9223372036854775807
2 A -  Requested: 9223372036854775807
3 A -  Received item: 0
4 A -  Received item: 1
5 B -  Received item: 0
6 A -  Received item: 2
7 A -  Received item: 3
8 B -  Received item: 2
```

Dropping excessive elements

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A - ⬇ Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A - ⬇ Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A - ⬆ Requested: " + l))
5   // Drop on overflow
6   .onOverflow().drop()
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Durat
8     .onItem().invoke(i -> System.out.println("B - ⬇ Received item: " + i))
9     .onFailure().invoke(f -> System.out.println("B - ⬇ Failed with " + f))
10    .onRequest().invoke(l -> System.out.println("B - ⬆ Requested: " + l))
11    .subscribe().with(__ -> {});
```

```
1 B - ⬆ Requested: 9223372036854775807
2 A - ⬆ Requested: 9223372036854775807
3 A - ⬇ Received item: 0
4 A - ⬇ Received item: 1
5 B - ⬇ Received item: 0
6 A - ⬇ Received item: 2
7 A - ⬇ Received item: 3
8 B - ⬇ Received item: 2
```

Buffering

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5   // Buffer on overflow
6   .onOverflow().buffer(10)
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Durat
8     .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
9     .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
10    .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
11    .subscribe().with(__ -> {});
```

```
B -  Requested: 9223372036854775807
A -  Requested: 9223372036854775807
A -  Received item: 0
A -  Received item: 1
B -  Received item: 0
A -  Received item: 2
B -  Received item: 1
A -  Received item: 3
B -  Received item: 2
```

Buffering

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5   // Buffer on overflow
6   .onOverflow().buffer(10)
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Duration.ofMillis(10)))
8   .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
9   .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
10  .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
11  .subscribe().with(__ -> {});
```

```
B -  Requested: 9223372036854775807
A -  Requested: 9223372036854775807
A -  Received item: 0
A -  Received item: 1
B -  Received item: 0
A -  Received item: 2
B -  Received item: 1
A -  Received item: 3
B -  Received item: 2
```

Buffering

```
1 Multi.createFrom().ticks().every(Duration.ofMillis(900))
2   .onItem().invoke(i -> System.out.println("A -  Received item: " + i))
3   .onFailure().invoke(f -> System.out.println("A -  Failed with " + f))
4   .onRequest().invoke(l -> System.out.println("A -  Requested: " + l))
5   // Buffer on overflow
6   .onOverflow().buffer(10)
7   .onItem().transformToUni(e -> Uni.createFrom().item(e).onItem().delayIt().by(Durat
8     .onItem().invoke(i -> System.out.println("B -  Received item: " + i))
9     .onFailure().invoke(f -> System.out.println("B -  Failed with " + f))
10    .onRequest().invoke(l -> System.out.println("B -  Requested: " + l))
11    .subscribe().with(__ -> {});
```

```
B -  Requested: 9223372036854775807
A -  Requested: 9223372036854775807
A -  Received item: 0
A -  Received item: 1
B -  Received item: 0
A -  Received item: 2
B -  Received item: 1
A -  Received item: 3
B -  Received item: 2
```

Server-Sent Events

Server-Sent Events is a technology that allows the server to push data to the client when it wants. The client opens a connection and the connection is kept open. The server can send chunks of data.

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Server-Sent Events

Here's an example of an endpoint that sends a chunk every second, containing the current time:

```
1 @GET
2 @Produces(MediaType.SERVER_SENT_EVENTS)
3 @RestSseElementType(MediaType.TEXT_PLAIN)
4 public Multi<String> time() {
5     return Multi.createFrom()
6         .ticks()
7         .every(Duration.ofSeconds(1))
8         .map(__ -> LocalDateTime.now().toString());
9 }
```

```
1 → lunatech-beginner-quarkus-course git:(main) ✘ http localhost:8082/time --stream
2 HTTP/1.1 200 OK
3 Content-Type: text/event-stream
4 X-SSE-Content-Type: text/plain
5 transfer-encoding: chunked
6 data:2021-02-23T14:09:59.233302
7 data:2021-02-23T14:10:00.237587
8 data:2021-02-23T14:10:01.236240
9 data:2021-02-23T14:10:02.236214
10 data:2021-02-23T14:10:03.236526
11 ^C
```

Reactive Messaging

Micropattern Reactive Messaging Spec

Micropattern Reactive Messaging specifies an annotation based model for creating event-driven microservices.

It has two main annotations:

- `@Incoming` to let a method *read* messages from a *channel*.
- `@Outgoing` to *write* the return values of a method to a *channel*.

Channels, Messages and Ack

Channels can be *internal*, to connect different beans to each other, or *external*, for example to connect them to Kafka. In Quarkus, this is managed by configuration.

Messages are the items that are produced by and written to Channels. It's a minimal interface that only contains methods to retrieve the payload, and to acknowledge the message.

To acknowledge the message means to tell the producer of the message that we've successfully processed it, and don't need to retrieve it again.

Internal Channels

```
1 @Outgoing("greet-subjects")
2 public Multi<String> greetSubjectsProducer() {
3     return Multi.createFrom()
4         .ticks()
5         .every(Duration.ofSeconds(1))
6         .map(__ -> faker.name().fullName())
7         .onOverflow().buffer(1);
8 }
9
10 @Incoming("greet-subjects")
11 @Outgoing("greets")
12 public String greetSubjectsConsumer(String subject) {
13     return "Hello " + subject + "!";
14 }
15
16 @Incoming("greets")
```

Internal Channels

```
1 @Outgoing("greet-subjects")
2 public Multi<String> greetSubjectsProducer() {
3     return Multi.createFrom()
4         .ticks()
5         .every(Duration.ofSeconds(1))
6         .map(__ -> faker.name().fullName())
7         .onOverflow().buffer(1);
8 }
9
10 @Incoming("greet-subjects")
11 @Outgoing("greets")
12 public String greetSubjectsConsumer(String subject) {
13     return "Hello " + subject + "!";
14 }
15
16 @Incoming("greets")
```

Internal Channels

```
4     .ticks()
5         .every(Duration.ofSeconds(1))
6         .map(__ -> faker.name().fullName())
7         .onOverflow().buffer(1);
8 }
9
10 @Incoming("greet-subjects")
11 @Outgoing("greets")
12 public String greetSubjectsConsumer(String subject) {
13     return "Hello " + subject + "!";
14 }
15
16 @Incoming("greets")
17 public void greetPrinter(String greet) {
18     System.out.println(greet);
19 }
```

Internal Channels

```
4     .ticks()
5     .every(Duration.ofSeconds(1))
6     .map(__ -> faker.name().fullName())
7     .onOverflow().buffer(1);
8 }
9
10 @Incoming("greet-subjects")
11 @Outgoing("greets")
12 public String greetSubjectsConsumer(String subject) {
13     return "Hello " + subject + "!";
14 }
15
16 @Incoming("greets")
17 public void greetPrinter(String greet) {
18     System.out.println(greet);
19 }
```

Internal Channels

```
1 @Outgoing("greet-subjects")
2 public Multi<String> greetSubjectsProducer() {
3     return Multi.createFrom()
4         .ticks()
5         .every(Duration.ofSeconds(1))
6         .map(__ -> faker.name().fullName())
7         .onOverflow().buffer(1);
8 }
9
10 @Incoming("greet-subjects")
11 @Outgoing("greets")
12 public String greetSubjectsConsumer(String subject) {
13     return "Hello " + subject + "!";
14 }
15
16 @Incoming("greets")
```

Signatures

Other signatures are supported as well:

```
1 @Incoming("greet-subjects")
2 @Outgoing("greets")
3 public Multi<String> greetSubjectsConsumer(String subject) {
4     return Multi.createFrom().items(
5         "Hey " + subject + "!",
6         "Ho " + subject + "!");
7 }
8
9 @Incoming("greets")
10 public CompletableFuture<Message<?>> greetPrinter(Message<String> greet) {
11     System.out.println(greet.getPayload());
12     return CompletableFuture.completedFuture(greet);
13 }
```

Signatures

Other signatures are supported as well:

```
1 @Incoming("greet-subjects")
2 @Outgoing("greets")
3 public Multi<String> greetSubjectsConsumer(String subject) {
4     return Multi.createFrom().items(
5         "Hey " + subject + "!",
6         "Ho " + subject + "!");
7 }
8
9 @Incoming("greets")
10 public CompletableFuture<Message<?>> greetPrinter(Message<String> greet) {
11     System.out.println(greet.getPayload());
12     return CompletableFuture.completedFuture(greet);
13 }
```

Signatures

Other signatures are supported as well:

```
1 @Incoming("greet-subjects")
2 @Outgoing("greets")
3 public Multi<String> greetSubjectsConsumer(String subject) {
4     return Multi.createFrom().items(
5         "Hey " + subject + "!",
6         "Ho " + subject + "!");
7 }
8
9 @Incoming("greets")
10 public CompletableFuture<Message<?>> greetPrinter(Message<String> greet) {
11     System.out.println(greet.getPayload());
12     return CompletableFuture.completedFuture(greet);
13 }
```

Signatures

Other signatures are supported as well:

```
1 @Incoming("greet-subjects")
2 @Outgoing("greets")
3 public Multi<String> greetSubjectsConsumer(String subject) {
4     return Multi.createFrom().items(
5         "Hey " + subject + "!",
6         "Ho " + subject + "!");
7 }
8
9 @Incoming("greets")
10 public CompletableFuture<Message<?>> greetPrinter(Message<String> greet) {
11     System.out.println(greet.getPayload());
12     return CompletableFuture.completedFuture(greet);
13 }
```

Signatures

All possibilities are specified in the MicroProfile Reactive Messaging Specification:

Methods consuming data

Signature	Behavior	Invocation
<pre>@Incoming("channel") Subscriber<Message<I>> method()</pre>	Returns a Subscriber that receives the Message objects transiting on the channel channel .	The method is called only once to retrieve the Subscriber object at assembly time. This subscriber is connected to the matching channel.
<pre>@Incoming("channel") Subscriber<I> method()</pre>	Returns a Subscriber that receives the <i>payload</i> objects transiting on the channel channel . The payload is automatically extracted from the inflight messages using Message.getPayload() .	The method is called only once to retrieve the Subscriber object at assembly time. This subscriber is connected to the matching channel.
<pre>@Incoming("channel") SubscriberBuilder<Message<I>> method()</pre>	Returns a SubscriberBuilder that receives the Message objects transiting on the channel channel .	The method is called only once at assembly time to retrieve a SubscriberBuilder that is used to build a CompletionSubscriber that is subscribed to the matching channel.

Exercise: Internal Channels

Connecting to Kafka

To connect to Kafka instead of through internal channels we need:

- The `quarkus-reactive-messaging-smallrye-kafka` extension
- Configure the Kafka Topics
- Configure the deserializer and serializer

Connecting to Kafka

```
1 mp.messaging.incoming.greets-in.connector=smallrye-kafka
2 mp.messaging.incoming.greets-in.topic=greets
3 mp.messaging.incoming.greets-in.value.deserializer=com.lunatech.training.quarkus.read
4 mp.messaging.outgoing.greets-out.connector=smallrye-kafka
5 mp.messaging.outgoing.greets-out.topic=greets
6 mp.messaging.outgoing.greets-out.value.serializer=io.quarkus.kafka.client.serializati
```

```
public class GreetDeserializer extends ObjectMapperDeserializer<Greet> {

    public GreetDeserializer() {
        super(Greet.class);
    }

}
```

Connecting to Kafka

```
1 mp.messaging.incoming.greets-in.connector=smallrye-kafka
2 mp.messaging.incoming.greets-in.topic=greets
3 mp.messaging.incoming.greets-in.value.deserializer=com.lunatech.training.quarkus.read
4 mp.messaging.outgoing.greets-out.connector=smallrye-kafka
5 mp.messaging.outgoing.greets-out.topic=greets
6 mp.messaging.outgoing.greets-out.value.serializer=io.quarkus.kafka.client.serializati
```

```
public class GreetDeserializer extends ObjectMapperDeserializer<Greet> {

    public GreetDeserializer() {
        super(Greet.class);
    }

}
```

Connecting to Kafka

```
1 mp.messaging.incoming.greets-in.connector=smallrye-kafka
2 mp.messaging.incoming.greets-in.topic=greets
3 mp.messaging.incoming.greets-in.value.deserializer=com.lunatech.training.quarkus.read
4 mp.messaging.outgoing.greets-out.connector=smallrye-kafka
5 mp.messaging.outgoing.greets-out.topic=greets
6 mp.messaging.outgoing.greets-out.value.serializer=io.quarkus.kafka.client.serializati
```

```
public class GreetDeserializer extends ObjectMapperDeserializer<Greet> {

    public GreetDeserializer() {
        super(Greet.class);
    }

}
```

Connecting to Kafka

```
1 mp.messaging.incoming.greets-in.connector=smallrye-kafka
2 mp.messaging.incoming.greets-in.topic=greets
3 mp.messaging.incoming.greets-in.value.deserializer=com.lunatech.training.quarkus.read
4 mp.messaging.outgoing.greets-out.connector=smallrye-kafka
5 mp.messaging.outgoing.greets-out.topic=greets
6 mp.messaging.outgoing.greets-out.value.serializer=io.quarkus.kafka.client.serializati
```

```
public class GreetDeserializer extends ObjectMapperDeserializer<Greet> {

    public GreetDeserializer() {
        super(Greet.class);
    }

}
```

Connecting to Kafka

```
1 mp.messaging.incoming.greets-in.connector=smallrye-kafka
2 mp.messaging.incoming.greets-in.topic=greets
3 mp.messaging.incoming.greets-in.value.deserializer=com.lunatech.training.quarkus.read
4 mp.messaging.outgoing.greets-out.connector=smallrye-kafka
5 mp.messaging.outgoing.greets-out.topic=greets
6 mp.messaging.outgoing.greets-out.value.serializer=io.quarkus.kafka.client.serializati
```

```
public class GreetDeserializer extends ObjectMapperDeserializer<Greet> {

    public GreetDeserializer() {
        super(Greet.class);
    }

}
```

Failure Strategies

Let's make our consumer crash occasionally:

```
1 private int counter = 0;
2
3 @Incoming("greets-in")
4 public void greetPrinter(Greet greet) {
5     if(++counter % 3 == 0) {
6         throw new RuntimeException("Crashing on message for " + greet.subject);
7     }
8
9     System.out.println(greet.greet + " " + greet.subject);
10 }
```

This will cause the stream to terminate on the third message.

Failure Strategies

Let's make our consumer crash occasionally:

```
1 private int counter = 0;
2
3 @Incoming("greets-in")
4 public void greetPrinter(Greet greet) {
5     if(++counter % 3 == 0) {
6         throw new RuntimeException("Crashing on message for " + greet.subject);
7     }
8
9     System.out.println(greet.greet + " " + greet.subject);
10 }
```

This will cause the stream to terminate on the third message.

Failure Strategies

Let's make our consumer crash occasionally:

```
1 private int counter = 0;
2
3 @Incoming("greets-in")
4 public void greetPrinter(Greet greet) {
5     if(++counter % 3 == 0) {
6         throw new RuntimeException("Crashing on message for " + greet.subject);
7     }
8
9     System.out.println(greet.greet + " " + greet.subject);
10 }
```

This will cause the stream to terminate on the third message.

Failure Strategies

We can configure this behaviour with:

```
mp.messaging.incoming.greets-in.failure-strategy=ignore
```

With this setting, it will log the error, but continue with the stream.

Failure Strategies

Another option:

```
mp.messaging.incoming.greets-in.failure-strategy=dead-letter-queue
```

This moves bad messages to a *dead letter queue*, where another system - or human operators - can inspect it, and maybe reschedule it.

Commit Strategies

Kafka is an *at-least-once* delivery system. To indicate to Kafka that you've successfully processed a message, you *commit* it. This will tell Kafka you don't need to see that particular message again. If you don't commit a message, then in case of a crash and a restart of your application, you will see the same message from Kafka again.

SmallRye Reactive Messaging Kafka keeps track of all acknowledgements of messages, and based on that decides when to commit.

Commit Strategies

Suppose the connector passed the following 8 messages to our app and received the following acknowledgements:

1. Acknowledged
2. Acknowledged
3. Acknowledged
4. Nothing
5. Acknowledged
6. Acknowledged
7. Nothing
8. Nothing

Now it can commit up to message #3, because everything up to message #3 has been acknowledged.

Commit Strategies

There are three commit strategies available:

- Ignore
- Latest
- Throttled

Exercise: Kafka

Exercise: Dead Letter Queue & Stream filtering

Non Functional Features

Learning outcomes

After this module, you should:

- Understand how to add health checks
- Understand the different approaches to gathering metrics
- Understand how to add tracing

Monitoring

- Task of / tools for observing our application over a period of time
 - Observing overall status and measuring behaviour
- Especially relevant in cloud-native (and K8s-native) context

Live**ness** and Readiness

- Quarkus Smallrye Health extension (implementation of MicroProfile Health)
- Adding extension gives out-of-the-box endpoints
 - /health
 - /health/live
 - /health/ready
- Liveness - Is the service *up/down, reachable/unreachable?*
- Readiness - Can the service handle user requests?
 - A service can be "UP" and pass the Liveness check but fail the Readiness check
- Correspond to liveness and readiness Kubernetes probes

Metrics (w / MicroProfile Metrics)

- Quarkus Smallrye Metrics extension (implementation of MicroProfile Metrics)
- Adding extension gives out-of-the-box endpoints
 - `/metrics`
 - `/metrics/base`
 - `/metrics/vendor`
 - `/metrics/application`
- Supports returning both JSON and OpenMetrics
 - OpenMetrics is a CNF sandbox project
 - OpenMetrics standard is based on Prometheus representation

Metrics (w / Micrometer)

- The recommended approach!
- Micrometer is the SLF4J for metrics
 - a vendor-neutral façade
- Quarkus Micrometer extension (with Prometheus registry)
- Adding extension gives out-of-the-box `/metrics` endpoint
 - Prometheus-formatted, with application metrics as well as system, jvm, and http metrics
 - JSON formatting can be enabled through config

Tracing

- Quarkus Smallrye OpenTracing (implementation of MicroProfile OpenTracing)
 - Uses Jaeger tracer
 - Trace IDs can be logged via MDC propagation
 - Can configure to trace JDBC requests and Kafka message deliver

Exercise: Observability (Bonus)

- Add in `application.properties`

```
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, parentId=%X{parentId}
```

- Add a JBoss Logger in `ProductsResource` and `PriceUpdatesResource` and add log messages for a few endpoints

```
import org.jboss.logging.Logger;  
  
private static final Logger LOGGER = Logger.getLogger(Foo.class)
```

- Add the following extensions:
 - Smallrye Health
 - Quarkus Micrometer
 - Quarkus Smallrye OpenTracing

Recap

In this module we have:

- Seen how to add liveness and readiness checks
- Seen two ways of enabling metrics gathering on our Quarkus app
- Seen how to enable distributed tracing

Conclusion

Resources

- Guides - <https://quarkus.io/guides/>
- Cheat Sheet - <https://lordofthejars.github.io/quarkus-cheat-sheet/>
- Blog - <https://quarkus.io/blog/>

Thank you