

# Lunatech Formation Quarkus

## Débutant

v1.0

Aperçu de la  
formation

# Bienvenue dans Quarkus pour Débutant par Lunatech

Au cours de cette formation vous apprendrez:

- La philosophie Quarkus
- L'injection de dépendances avec ArC
- La couche d'accès aux données de Quarkus
- Le moteur de template Qute
- Le développement d'APIs RESTful
- La programmation Réactive
- La surveillance des applications Quarkus
- Tester son application
- ... et de nombreux outils et bonnes pratiques Quarkus

# Pré-requis

Vous devez préalablement avoir :

- Connaissance du langage Java
- Connaissances basiques du Web et de HTTP

Votre machine devrait être installée avec:

- Java 11
- Docker
- L'IDE de votre choix

# Planning jour 1

## Matin

- Démarrage
- Moteur de template Qute
- Accès aux bases de données

## Après-midi

- Web Services
- OpenAPI
- Test

# Planning jour 2

## Matin

- Modèle d'exécution de Quarkus
- Programmation réactive
- Messagerie réactive

## Après-midi

- Programmation réactive (suite)
- Sujets optionnels (si le temps le permet)
  - Observabilité
  - Serverless avec AWS

# Etude de cas

Nous allons réaliser plusieurs exercices construits autour d'un même projet : un magasin de meubles nommé HIQUÉA.

A propos des formateurs(s)

# A propos des participants

# Informations pratiques

- Horaires: 09h30 - 17h30
- GitHub Repo

# Démarrage

# Connaissances obtenues

A l'issue de ce module, vous devriez :

- Comprendre la philosophie et les principes fondamentaux de Quarkus
- Etre capable de générer un nouveau projet Quarkus depuis le site web Quarkus
- Savoir comment démarrer Quarkus depuis la ligne de commande en utilisant le wrapper Maven
- Savoir comment accéder à l'interface de développement

# Pourquoi Quarkus existe

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

*Supersonic Subatomic Java ... Une stack Java nativement Kubernetes... construite à partir de la sélection des meilleures librairies et standards Java.*

<https://quarkus.io/>

# Pourquoi Quarkus existe

- *Java à l'ère de l'infonuagique*
- Unifie les modèles impératif et réactif
- Satisfaction des développeurs

# Pourquoi Quarkus existe

- Java à l'ère de l'infonuagique
- *Unifie les modèles impératif et réactif*
- Satisfaction des développeurs

# Pourquoi Quarkus existe

- Java à l'ère de l'infonuagique
- Unifie les modèles impératif et réactif
- *Satisfaction des développeurs*

# Construit sur des standards, inspiré des meilleures pratiques

- Quarkus et de nombreuses extensions sont bâties sur des standards de l'industrie tels que Jakarta EE et MicroProfile
- Quarkus lui même est construit sur un des meilleurs frameworks réactifs, Vert.x
- Quarkus est inspiré par l'expérience développeur d'autres frameworks comme Spring Boot et Play Java
  - Quarkus possède même une extension de compatibilité avec Spring
- Quarkus implémente de nombreuses APIs et supporte beaucoup de frameworks. Cela fait partie de la philosophie de Quarkus.

# Démo Hello World

# Hello World en Quarkus

Nous pouvons utiliser les annotations JAX-RS pour créer un point d'entrée “Hello World”:

```
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 est une spécification de Jakarta EE API. Il contient des annotations telles que :

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

En somme, une manière standardisée de définir des web services  
RESTful

# JAX-RS & RESTeasy

RESTEasy est l'implémentation RedHat du standard JAX-RS. C'est ce que Quarkus utilise pour fournir des web services.

# Exercice: Hello World

# Récapitulatif

Dans ce module, nous avons :

- Discuté de la philosophie de Quarkus
- Mis en place un projet de base qui sera utilisé pour la suite
- Découvert la joie des développeurs que Quarkus cherche à déclencher

Qute

# Connaissances obtenues

A l'issue de ce module, vous devriez :

- Comprendre les motivations derrières Qute
- Apprendre à créer et utiliser une template Qute
- Apprendre à créer des tags Qute personnalisés et des extensions de méthodes

# Moteur de template *Qute*

Quarkus est livré avec un moteur de template appelé *Qute* (**Quarkus templating**):

- Une syntaxe simple
- Minimiser l'usage de la réflexion
- Optionnellement, sûreté du typage
- La sortie peut être streamée

# 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

# Les expressions de Qute

Avec une classe `Product` donné:

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

C'est une template présentant les détails d'un produit :

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

# Iterations avec Qute

Vous pouvez faire l'itération d'une collection:

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

# Quelques opérateurs Qute

Qute possède quelques opérateurs utiles:

```
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}
```

# Quelques opérateurs Qute

Qute possède quelques opérateurs utiles:

```
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}
```

# Quelques opérateurs Qute

Qute possède quelques opérateurs utiles:

```
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}
```

# Quelques opérateurs Qute

Qute possède quelques opérateurs utiles:

```
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}
```

# Quelques opérateurs Qute

Qute possède quelques opérateurs utiles:

```
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}
```

# Quelques opérateurs Qute

Qute possède quelques opérateurs utiles:

```
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}
```

# Quelques opérateurs Qute

Qute possède quelques opérateurs utiles:

```
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}
```

# Utilisation de Qute

```
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. Injection d'un template. Quarkus dérive le nom du fichier du template file à partir du nom du champ.
2. La méthode dans la ressource retourne un `TemplateInstance`. RESTeasy sait comment le convertir en réponse HTML.
3. Alimente le template avec les données pour créer un `TemplateInstance`.

# Utilisation de Qute

```
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. Injection d'un template. Quarkus dérive le nom du fichier du template file à partir du nom du champ.
2. La méthode dans la ressource retourne un `TemplateInstance`. RESTeasy sait comment le convertir en réponse HTML.
3. Alimente le template avec les données pour créer un `TemplateInstance`.

# Utilisation de Qute

```
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. Injection d'un template. Quarkus dérive le nom du fichier du template file à partir du nom du champ.
2. La méthode dans la ressource retourne un `TemplateInstance`. RESTeasy sait comment le convertir en réponse HTML.
3. Alimente le template avec les données pour créer un `TemplateInstance`.

# Utilisation de Qute

```
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. Injection d'un template. Quarkus dérive le nom du fichier du template file à partir du nom du champ.
2. La méthode dans la ressource retourne un `TemplateInstance`. RESTeasy sait comment le convertir en réponse HTML.
3. Alimente le template avec les données pour créer un `TemplateInstance`.

# Utilisation de Qute

```
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. Injection d'un template. Quarkus dérive le nom du fichier du template file à partir du nom du champ.
2. La méthode dans la ressource retourne un `TemplateInstance`. RESTeasy sait comment le convertir en réponse HTML.
3. Alimente le template avec les données pour créer un `TemplateInstance`.

Exercice: Un Hello World avec  
Qute

# Méthodes virtuelles Qute

Qute autorise l'appel de méthodes virtuelles sur des valeurs. Elles sont appelées virtuelles car elles ne correspondent pas à de réelles méthodes dans l'objet Java:

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

# Méthodes virtuelles Qute

Qute autorise l'appel de méthodes virtuelles sur des valeurs. Elles sont appelées virtuelles car elles ne correspondent pas à de réelles méthodes dans l'objet Java:

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

# Méthodes virtuelles Qute

Qute autorise l'appel de méthodes virtuelles sur des valeurs. Elles sont appelées virtuelles car elles ne correspondent pas à de réelles méthodes dans l'objet Java:

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

# Méthodes virtuelles Qute

```
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>
```

# Méthodes virtuelles Qute

```
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>
```

# Méthodes virtuelles Qute

```
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>
```

# Méthodes virtuelles Qute

```
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>
```

# Méthodes virtuelles Quie

```
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>
```

# Méthodes virtuelles Quie

```
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>
```

# Méthodes virtuelles Quie

```
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>
```

# Méthodes virtuelles Qute

```
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>
```

# Méthodes virtuelles Quite

Nous ne pouvons appeler des méthodes *réelles* avec des paramètres:

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

Affichera:

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

# Méthodes virtuelles Qute - Template Data

Mais nous pouvons apprendre à Qute à générer un *value resolver* pour nous:

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

Maintenant cela fonctionne comme attendu:

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

# Templates à typage-sûr

Dans l'exemple précédent, nous avons vu la ligne suivante

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

affichait NOT\_FOUND, lors de l'exécution. Nous pouvons améliorer cela, et faire en sorte que Qute génère une erreur à l'exécution, en indiquant dans le template que la valeur attendue est du 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>
```

# Templates à typage-sûr

Dans l'exemple précédent, nous avons vu la ligne suivante

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

affichait NOT\_FOUND, lors de l'exécution. Nous pouvons améliorer cela, et faire en sorte que Qute génère une erreur à l'exécution, en indiquant dans le template que la valeur attendue est du 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>
```

# Templates à typage-sûr

Dans l'exemple précédent, nous avons vu la ligne suivante

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

affichait NOT\_FOUND, lors de l'exécution. Nous pouvons améliorer cela, et faire en sorte que Qute génère une erreur à l'exécution, en indiquant dans le template que la valeur attendue est du 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>
```

# Templates à typeage-sûr

Maintenant, Qute devrait afficher une erreur:

## 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>
```

# Exercice: Produits avec Qute, partie 1

Exercice: Produits avec Qute,  
partie 2

# Récapitulatif

Dans ce module nous avons:

- Discuté pourquoi Qute a été créé et comment il est différent des autres moteurs de templating
- Crée un template Qute et utilisé celui-ci à partir d'une Ressource
- Vu comment créer des tags personnalisés et des extensions de méthodes
- Vu comment créer des templates à typage-sûr

# Persistence

# Connaissances obtenues

A l'issue de ce module, vous devriez :

- Comprendre les différentes options de connectivité
- Savoir configurer une source de données
- Savoir utiliser Hibernate + Panache pour récupérer et stocker des données
- Comprendre où l'annotation `@Transactional` peut être placée

# Multiples options pour la persistance

- Hibernate ORM et JPA
- Hibernate ORM avec Panache
- Reactive SQL
- Plusieurs clients NoSQL (MongoDB, Redis, Neo4j, Cassandra, etc.)

# Quarkus, Hibernate & Panache

- Quarkus et Hibernate sont les meilleurs copains.
- Panache est une couche qui :
  - Facilite les *Active Record* ou *Repository patterns*
  - ... avec de nombreuses méthodes pré-crées
  - Peut créer des points d'extrémité RESTful pour les entités



# 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=...
```

# Exemple d'un *Active Record*

```
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(){}
```

# Exemple d'un *Active Record*

```
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(){}
```

# Exemple d'un *Active Record*

```
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(){
17         delete("name = 'Chair'")
```

# Exemple d'un *Active Record*

```
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 }
```

# Exemple d'un *Active Record*

```
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 }
```

# Exemple d'un *Active Record*

```
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(){}
```

# Exemple pour un *Repository*

```
@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");
    }
}
```

# Pagination

```
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))
```

# Pagination

```
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))
```

# Pagination

```
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))
```

# Pagination

```
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))
```

# Pagination

```
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))
```

# Pagination

```
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))
```

# Pagination

```
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))
```

# Pagination

```
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))
```

# Pagination

```
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))
```

# Triage

```
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"));
}
```

# Requête de 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
```

# Réécriture de l'accès aux champs

- Vous pouvez écrire vos Entités Panache avec des champs publics
- Quarkus réécrira automatiquement tous les accès aux getters et setters (générés)
- Vous pouvez réécrire les getters and setters quand vous le souhaitez.

# Exemple de réécriture de l'accès aux champs

```
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);
```

# Exemple de réécriture de l'accès aux champs

```
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);
```

# Exemple de réécriture de l'accès aux champs

```
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

- Toutes les extensions liées à la persistance intègrent le *Transaction Manager*
- Une approche declarative avec l'annotation `@Transactional`
- Six types de configuration
  - REQUIRED (la valeur par défaut)
  - REQUIRED\_NEW
  - MANDATORY
  - SUPPORTS
  - NOT\_SUPPORTED
  - NEVER

# Insérer nos produits dans la base de données

En mode développement:

- `import.sql` dans le répertoire `src/main/resources`
- `quarkus.hibernate-orm.database.generation=drop-and-create`

En production on utiliserait plutôt la *schema migration* avec Flyway

Exercice : Produit depuis la base  
de données

# CDI & ArC

CDI est une spécification de Jakarta EE et MicroProfile pour l'injection  
de contexte et de dépendance

Quarkus possède une implémentation partielle de *ArC*

# ArC - Temps pour build avec DI (injection de dépendance)

- Au moment de la compilation, ArC analyse toutes les classes et dépendances
- Au moment de l'exécution, ArC n'a plus qu'à lire les métadonnées générées et à instancier les classes.

# Fonctionnalités

- Champs, constructeur et setter injection
- @Dependent, @ApplicationScoped, @Singleton, @RequestScoped et @SessionScoped *scopes*
- @AroundInvoke, @PostConstruct, @PreDestroy, @AroundConstruct *lifecycle callbacks* et *interceptors*

# Injection

- En gros, ce à quoi vous vous attendez
- Utiliser l'annotation `@Inject`
- Tout est résolu durant la compilation, il n'y a donc pas de `@Conditional` comme avec Spring
- Mais il y a `@IfBuildProperty`

# Scopes (Portées)

- Portées normales: `@ApplicationScoped`, `@RequestScoped`,  
`@SessionScoped` - Crée lorsqu'une méthode est invoquée.
- Pseudo-portées: `@Singleton`, `@Dependent` - Crée lorsqu'il est injecté.

# Lifecycle callbacks

```
@ApplicationScoped
public class MyBean {

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

# Interceptors

## Créer un *Interceptor Binding*

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

Et un *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

## Créer un *Interceptor Binding*

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

Et un *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

## Créer un *Interceptor Binding*

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

Et un *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

## Créer un *Interceptor Binding*

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

Et un *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

## Créer un *Interceptor Binding*

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

Et un *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

Maintenant nous pouvons l'utiliser sur une méthode:

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

On pourrait aussi le mettre sur la classe, pour chronométrer toutes les invocations de méthode sur la classe.

# Caractéristiques non standard

- `@Inject` peut être ignoré si une annotation comme `@ConfigProperty` est présente
- Les constructeurs sans arguments peuvent être ignorés, et `@Inject` n'est pas nécessaire s'il n'y a qu'un seul constructeur
- Marquer un bean destiné à être réécrit avec `@DefaultBean`

# Exercise: CDI & ArC

# Recapitulatif

Après ce module, vous avez :

- Configuré une source de données Quarkus pour se connecter à PostgreSQL
- Ajouté l'extension Hibernate+Panache pour notre couche de persistance
- Vu comment utiliser l'annotation `@Transactional`
- Exploré l'injection de dépendance avec CDI et ArC

# Web Services

# Connaissances obtenues

Après ce module, vous devrez :

- Savoir écrire un endpoint GET qui retourne un JSON
- Savoir écrire un endpoint POST qui prend et valide un JSON
- Savoir comment générer une spécification Open API
- Savoir comment utiliser `@QuarkusTest` pour écrire un test d'intégration pour les endpoints RESTful

# Introduction

Dans ce chapitre, nous allons mettre à jour nos endpoints Qute en endpoints JSON, et interagir avec eux en utilisant un frontend React.

# Sérialisation automatique en JSON

Donnant une classe `Greet`:

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

# Sérialisation automatique JSON

Quarkus et RESTeasy peuvent automatiquement sérialiser ceci en JSON,  
si nous lui indiquons de produire du JSON:

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

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

Le retour:

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

# Sérialisation automatique JSON avec Jackson

Nous pouvons utiliser les annotations Jackson pour modifier le JSON généré:

```
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 }
```

Maintenant, nous avons ce résultat:

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

# Sérialisation automatique JSON avec Jackson

Nous pouvons utiliser les annotations Jackson pour modifier le JSON généré:

```
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 }
```

Maintenant, nous avons ce résultat:

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

# Alternatives

Mais Quarkus n'est pas forcément lié à Jackson! Ici c'est le cas, car nous avons installé l'extension `quarkus-resteasy-jackson`. Mais si nous préférons JSON-B à la place (faisant partie de Microprofile!), nous pouvons utiliser l'extension `quarkus-resteasy-jsonb`. Et alors nous pouvons appliquer les annotations JSON-B:

```
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

Nous pouvons aussi avoir les deux extensions installées. Quarkus choisira l'implémentation du sérialiseur en se basant sur les annotations utilisées dans la classe.

# Jackson JSON object

Afin de retourner du JSON, nous pouvons utiliser la librairie Jackson:

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

Exercice: Convertir les endpoints  
en 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

# Exercice: Ajouter Open API

# Exercice: Ajouter REST data Panache

# Exercice: Tester ses endpoints

Exercice: 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;
}
```

Exercice: Créer un endpoint PUT,  
activer le feature-flag et essayer  
dans l'application react

# Recap

Au cours de ce module nous avons:

- Vu comment fonctionne la sérialisation JSON dans Quarkus
- Ajouté un point d'entrée POST avec la validation du contenu de la requête JSON
- Généré une spécification OpenAPI et vus comment accéder à l'interface Swagger
- Câblé un frontal React à notre application HIQUEA
- Vu comment utiliser la *Bean Validation* pour valider des requêtes vers nos endpoints REST

# Programmation réactive

# Modèle d'exécution

Lors de l'utilisation de RESTEasy, *impératif* par défaut, Quarkus crée autant de **executor threads** que nécessaire, jusqu'à atteindre le maximum configuré:

```
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
```

La configuration par défaut du maximum est `max(200, 8 * nr_of_cores)`

# Modèle d'exécution

Lors de l'utilisation de RESTEasy, *impératif* par défaut, Quarkus crée autant de **executor threads** que nécessaire, jusqu'à atteindre le maximum configuré:

```
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
```

La configuration par défaut du maximum est `max(200, 8 * nr_of_cores)`

# Modèle d'exécution

Lors de l'utilisation de RESTEasy, *impératif* par défaut, Quarkus crée autant de **executor threads** que nécessaire, jusqu'à atteindre le maximum configuré:

```
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
```

La configuration par défaut du maximum est `max(200, 8 * nr_of_cores)`

# Modèle d'exécution

Lors de l'utilisation de RESTEasy, *impératif* par défaut, Quarkus crée autant de **executor threads** que nécessaire, jusqu'à atteindre le maximum configuré:

```
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
```

La configuration par défaut du maximum est `max(200, 8 * nr_of_cores)`

# Modèle d'exécution

Lors de l'utilisation de RESTEasy, *impératif* par défaut, Quarkus crée autant de **executor threads** que nécessaire, jusqu'à atteindre le maximum configuré:

```
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
```

La configuration par défaut du maximum est `max(200, 8 * nr_of_cores)`

# Modèle d'exécution

Lors de l'utilisation de RESTEasy, *impératif* par défaut, Quarkus crée autant de **executor threads** que nécessaire, jusqu'à atteindre le maximum configuré:

```
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
```

La configuration par défaut du maximum est `max(200, 8 * nr_of_cores)`

# Modèle d'exécution

Si nous limitons volontairement le nombre maximal de fils (threads):

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

Alors l'exécution de la même commande ab prend beaucoup plus longtemps:

```
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
```

# Modèle d'exécution

Si nous limitons volontairement le nombre maximal de fils (threads):

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

Alors l'exécution de la même commande ab prend beaucoup plus longtemps:

```
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
```

# Modèle d'exécution

Si nous limitons volontairement le nombre maximal de fils (threads):

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

Alors l'exécution de la même commande ab prend beaucoup plus longtemps:

```
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
```

# Modèle d'exécution - Threads bloquants

Deux choses qui peuvent occuper un thread :

- Effectuer une tâche utile sur le CPU
- Attendre quelqu'un d'autre (Base de données, Appel d'API, IO Disque, etc.). C'est ce que l'on appelle (*bloquage*) *blocking*.

# Modèle d'exécution - Threads bloquants

Deux choses qui peuvent occuper un thread :

- Effectuer une tâche utile sur le CPU
- Attendre quelqu'un d'autre (Base de données, Appel d'API, IO Disque, etc.). C'est ce que l'on appelle (*bloquage*) *blocking*.

# RESTEasy Reactive

L'extension `quarkus-resteasy-reactive` ajoute un support du modèle réactif de JAX-RS à Quarkus.

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

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

Par exemple la définition du point d'entrée REST fonctionne exactement comme la version impérative.

# RESTEasy Reactive

L'extension `quarkus-resteasy-reactive` ajoute un support du modèle réactif de JAX-RS à 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 }
```

Mais nous pouvons également retourner un `CompletionStage`

# RESTEasy Reactive

L'extension `quarkus-resteasy-reactive` ajoute un support du modèle réactif de JAX-RS à 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 }
```

Mais nous pouvons également retourner un `CompletionStage`

# RESTEasy Reactive

L'extension `quarkus-resteasy-reactive` ajoute un support du modèle réactif de JAX-RS à 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 }
```

Mais nous pouvons également retourner un `CompletionStage`

# Le modèle d'exécution réactif

RESTEasy reactive n'est pas indifférent aux bloquages

- Votre méthode sera appelée par le thread de la boucle d'évènement Vert.x (eventloop)
- Vous ne devriez pas le bloquer
- Mais si vous devez, annoter avec `@Blocking`

# Le modèle d'exécution réactif

## Exemple

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

C'est bien - retourne le nom du thread de la boucle Vert.x, par exemple  
`vert.x-eventloop-thread-3`

# Le modèle d'exécution réactif

## Mauvais exemple

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

Ce n'est pas correct.

# Le modèle d'exécution réactif

## Mauvais exemple

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

Ce n'est pas correct.

# Le modèle d'exécution réactif

## Mauvais exemple

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

Ce n'est pas correct.

# Le modèle d'exécution réactif

## Mauvais exemple

```
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.
```

# Le modèle d'exécution réactif

## Mauvais exemple

```
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.
```

# Le modèle d'exécution réactif

## Mauvais exemple

```
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.
```

# Le modèle d'exécution réactif

## Mauvais exemple

```
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.
```

# Le modèle d'exécution réactif

## Mauvais exemple

```
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.
```

# Le modèle d'exécution réactif

## Bon exemple

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

Ceci retourne `executor-thread-221`

# Le modèle d'exécution réactif

## Bon exemple

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

Ceci retourne `executor-thread-221`

# Le modèle d'exécution réactif

## Bon exemple

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

Ceci retourne `executor-thread-221`

# Le modèle d'exécution réactif

## Bon exemple

```
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
```

Retour à des performances normales :)

# Le modèle d'exécution réactif

## Un exemple encore meilleur

Evidemment, nous pouvons faire encore mieux, en ne bloquant pas du tout le thread :

```
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 }
```

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

# Le modèle d'exécution réactif

## Un exemple encore meilleur

Evidemment, nous pouvons faire encore mieux, en ne bloquant pas du tout le thread :

```
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 }
```

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

# Le modèle d'exécution réactif

## Un exemple encore meilleur

Evidemment, nous pouvons faire encore mieux, en ne bloquant pas du tout le thread :

```
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 }
```

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

# Le modèle d'exécution réactif

## Un exemple encore meilleur

Evidemment, nous pouvons faire encore mieux, en ne bloquant pas du tout le thread :

```
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 }
```

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

# Le modèle d'exécution réactif

## Un exemple encore meilleur

Evidemment, nous pouvons faire encore mieux, en ne bloquant pas du tout le thread :

```
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 }
```

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

# Reactive Routes

Une alternative à *RESTEasy Reactive* est d'utiliser l'extension *Reactive Routes*:

*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 }
```

Accès réactif à la  
base de données

# A propos de JDBC

JDBC est une API bloquante

Exemple:

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

Il n'existe pas de solution pour obtenir le **ResultSet** sans thread bloquante.

# Hibernate devient Réactif

Depuis Décembre 2020, Hibernate Reactive a été lancé:

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

C'est une API réactive pour Hibernate ORM.

# Hibernate devient Réactif

- Fonctionne avec les clients non-bloquant d'accès aux bases de données. Pour le moment, les clients Vert.x pour Postgres, MySQL et DB2
- L'intégration avec Quarkus est agréable
- Pas de *lazy* chargement bloquant, mais des opérations asynchrones explicites pour récupérer les associations

# Hibernate Reactive + Panache

- Les méthodes qui retournent T ou List<T> retourne maintenant Uni<T> et Uni<List<T>>
- Nouvelles méthodes streamXXX qui retournent Multi<T>
- Les classes sont dans un nouveau package  
`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** est la librairie pour la programmation réactive que Quarkus utilise. Il y a deux types principaux:

- `Multi<T>` représente un stream d'éléments de type T
- `Uni<T>`, représente un stream de zéro ou un élément de type T

# RESTEasy Reactive with Mutiny Uni

Les **Unis** sont supportés comme un type de résultat:

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

# RESTEasy Reactive with Mutiny Uni

Les **Unis** sont supportés comme un type de résultat:

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

# RESTEasy Reactive with Mutiny Uni

Les **Unis** sont supportés comme un type de résultat:

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

# RESTEasy Reactive

Multi est aussi supporté:

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

Cela retourne une réponse HTTP chunked.

# Exercise: Going Reactive

# Sessions & Transactions

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

# Sessions & Transactions - Exemple

Bon exemple:

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

Mauvais exemple:

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

Les méthodes:

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

Les deux exemples compilent et possèdent les types correctes, but le deuxième *déclenchera jamais l'opération flush.*

# Clients SQL Réactifs Low-level

Une autre possibilité pour se connecter à la base de données est d'utiliser un client SQL low-level.

```
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);
```

L'objet de base dont nous avons besoin est une instance de PgPool.

# Clients SQL Réactifs Low-level

```
@Inject
```

```
PgPool client;
```

Récupérer le bon PgPool:

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

# Requêteage

Les requêtes retournent un `Uni` contenant un `RowSet`:

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

Bien sûr, nous pouvons transformer cela par un `Multi` de `Rows`:

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

# Requêteage

```
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"));
}
```

# Paramètres

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

# Paramètres

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

# Paramètres

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

# Paramètres

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

# Paramètres

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

# Insertions et Mise à jour

```
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 }
```

# Insertions et Mise à jour

```
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 }
```

# Insertions et Mise à jour

```
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 }
```

# Insertions et Mise à jour

```
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 }
```

# Insertions et Mise à jour

```
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 }
```

# Insertions et Mise à jour

```
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

L'une des propriétés sympa de Postgres est d'écouter (`Listen`) des canaux. Dans le cadre des transactions, vous pouvez notifier les canaux, par exemple pour alerter les consommateurs qui attendent un événement.

# 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

Les flux de données (*Streaming data*) sont fréquents dans les applications modernes:

- Flux d'évènements d'un système vers des consommateurs
- Flux d'enregistrements d'une base de données vers une réponse HTTP chunked (découpée en gros morceaux)
- Messages consommés depuis une queue, transformés et déposés dans une autre queue

# Reactive Streams

Les flux de données (*Streaming data*) sont fréquents dans les applications modernes:

- Flux d'évènements d'un système vers des consommateurs
- Flux d'enregistrements d'une base de données vers une réponse HTTP chunked (découpée en gros morceaux)
- Messages consommés depuis une queue, transformés et déposés dans une autre queue

Un problème fondamental des systèmes de traitement de flux de données est d'avoir l'assurance que le *consommateur* du flux peut traiter les messages qu'on lui envoie.

# Consommateur lent

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?

# Consommateur lent - solutions

Solutions possibles:

# Consommateur lent - solutions

Solutions possibles:

- Avoir un consommateur réellement rapide

# Consommateur lent - solutions

Solutions possibles:

- Avoir un consommateur réellement rapide
- Avoir un producteur lent

# Consommateur lent - solutions

Solutions possibles:

- Avoir un consommateur réellement rapide
- Avoir un producteur lent
- Avoir plus de mémoire que la taille de votre base de données

# Consommateur lent - solutions

Solutions possibles:

- Avoir un consommateur réellement rapide
- Avoir un producteur lent
- Avoir plus de mémoire que la taille de votre base de données
- Adapter la vitesse du producteur, basé sur la capacité du consommateur

# Back pressure

La *Back pressure* signifie que le consommateur peut faire la *demande* au producteur. Le producteur ne produira que la quantité demandée par le consommateur.

# Back pressure

- Fonctionne avec le flux entier, et pas juste avec le consommateur à la fin
- Chaque élément peut adapter la demande qui a été envoyée en amont
  - Les composants lents réduisent la demande
  - Quelques composants, comme les buffers, peuvent augmenter la demande

# Les flux au dessus TCP

- TCP supporte nativement la back-pressure!
- Les messages *ack* (*accusés*) contiennent un champ *window* (*fenêtre*), indiquant comment l'expéditeur doit faire l'envoi.
- Quand le destinataire reçoit les données, un nouvel accusé est envoyé, avec une fenêtre plus grande.

# Les flux réactifs standards

A partir de 2013, les ingénieurs de Netflix, Pivotal, Lightbend, Twitter et d'autres ont tous travaillés autour des systèmes de flux, essentiellement en résolvants les mêmes problèmes.

Pour être certain que leurs librairies seront compatibles, ils ont mis en place un standard de **Reactive Streams**.

C'est une interface minimale nécessaire pour connecter les librairies de flux, retenant toutes les opérations non bloquantes, la back-pressure, le partage de l'annulation et le modèle d'erreur.

# Les flux réactifs standards

Terminé dans la bibliothèque standard Java à partir de Java 9, sous  
`java.util.concurrent.Flow`.

# Les implémentations de flux réactif

- Akka Streams
- RxJava
- Reactor
- Vert.x
- Mutiny
- ... et bien d'autres

# Démarrage

```
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);
```

# Démarrage

```
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);
```

# Démarrage

```
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);
```

# Démarrage

```
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

Un Uni n'est exécuté que lorsqu'il est connecté à un abonné (*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

Un Uni n'est exécuté que lorsqu'il est connecté à un abonné (*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

L'interface `Multi` de Mutiny étend  
`org.reactivestreams.Publisher<T>`, donc c'est un flux réactif.

Multi possède de nombreuse méthode pour interagir avec:

```
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

L'interface `Multi` de Mutiny étend  
`org.reactivestreams.Publisher<T>`, donc c'est un flux réactif.

Multi possède de nombreuse méthode pour interagir avec:

```
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

L'interface `Multi` de Mutiny étend  
`org.reactivestreams.Publisher<T>`, donc c'est un flux réactif.

Multi possède de nombreuse méthode pour interagir avec:

```
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

L'interface `Multi` de Mutiny étend  
`org.reactivestreams.Publisher<T>`, donc c'est un flux réactif.

Multi possède de nombreuse méthode pour interagir avec:

```
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

L'interface `Multi` de Mutiny étend  
`org.reactivestreams.Publisher<T>`, donc c'est un flux réactif.

Multi possède de nombreuse méthode pour interagir avec:

```
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

Un abonné (subscriber) à un Multi, doit faire face aux situations suivantes:

- Un élément arrive
- Une erreur intervient
- Le flux (borné) et complété

```
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

Un abonné (subscriber) à un Multi, doit faire face aux situations suivantes:

- Un élément arrive
- Une erreur intervient
- Le flux (borné) et complété

```
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

Un abonné (subscriber) à un Multi, doit faire face aux situations suivantes:

- Un élément arrive
- Une erreur intervient
- Le flux (borné) et complété

```
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

Un abonné (subscriber) à un Multi, doit faire face aux situations suivantes:

- Un élément arrive
- Une erreur intervient
- Le flux (borné) et complété

```
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"));
```

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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
```

L'affichage est le suivant:

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

# Visualisation des événements

```
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());
```

Cela affiche:

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

# Visualisation des événements

```
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());
```

Cela affiche:

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

# Visualisation des événements

```
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());
```

Cela affiche:

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

# Visualisation des événements

Nous pouvons configurer le nombre d'élément dans la 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
```

# Visualisation des événements

Nous pouvons configurer le nombre d'élément dans la 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
```

# Visualisation des événements

Nous pouvons configurer le nombre d'élément dans la 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
```

# Visualisation des événements

Nous pouvons configurer le nombre d'élément dans la 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
```

# Visualisation des événements

Le flux créé par `asStream` devrait annuler (*cancel*) le `Multi` quand il est fermé (`close`).

```
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
```

# Visualisation des événements

Le flux créé par `asStream` devrait annuler (*cancel*) le `Multi` quand il est fermé (`close`).

```
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
```

# Visualisation des événements

Le flux créé par `asStream` devrait annuler (*cancel*) le `Multi` quand il est fermé (`close`).

```
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
```

# Stratégies de Backpressure

Principalement, il y a trois choses à faire quand le producteur va plus vite que le consommateur:

- Réduire la vitesse du producteur
- Bufferiser les éléments
- Jeter des éléments

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Stratégies de Backpressure

```
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(__ -> {});`
```

Affiche:

```
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
```

# Rejeter les éléments en trop

```
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
```

# Rejeter les éléments en trop

```
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
```

# Rejeter les éléments en trop

```
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
```

# Rejeter les éléments en trop

```
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
```

# Rejeter les éléments en trop

```
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
```

# Rejeter les éléments en trop

```
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
```

# Rejeter les éléments en trop

```
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
```

# Bufferisé

```
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
```

# Bufferisé

```
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
```

# Bufferisé

```
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
```

# L'envoi d'événement depuis le serveur (Server-Sent Events)

L'envoi d'événement depuis le serveur (Server-Sent Events) est une technologie qui autorise le serveur à pousser des données à un client quand il le veut. Le client ouvre une connexion et la garde ouverte. Le serveur peut envoyer des données tronquées (chunks).

# Server-Sent Events

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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

Ci-dessous un exemple d'un endpoint qui envoi des données tronquées toute les secondes, contenant la date et heure courante:

```
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 spécifie une annotation basée sur le modèle pour la création de microservice événementiel.

Il y a deux annotations principales:

- `@Incoming` pour permettre à une méthode de *lire* les messages à partir d'un *canal*.
- `@Outgoing` pour *écrire* la valeur de retour d'une méthode dans un *canal*.

# Canaux, Messages et Accusé

Les canaux peuvent être *internes*, pour connecter différentes instances entre elles, ou *externes*, par exemple pour se connecter à Kafka. Dans Quarkus, cela est géré par la configuration.

Les messages sont les éléments produits et écrits sur les canaux. C'est une interface minimale qui ne contient que des méthodes pour récupérer le contenu et pour accuser réception du message.

Accuser réception du message signifie dire au producteur du message que nous l'avons traité avec succès et qu'il n'est pas nécessaire de le récupérer à nouveau.

# Canaux internes

```
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")
```

# Canaux internes

```
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")
```

# Canaux internes

```
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 }
```

# Canaux internes

```
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 }
```

# Canaux internes

```
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

D'autres signatures sont supportées:

```
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

D'autres signatures sont supportées:

```
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

D'autres signatures sont supportées:

```
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

D'autres signatures sont supportées:

```
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

Toute les possibilités sont spécifiées dans la spécification de messagerie réactive de Microprofile:

## Methods consuming data

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

# Exercise: Internal Channels

# Connexion à Kafka

Pour se connecter à Kafka au lieu de passer par des canaux internes,  
nous avons besoin de:

- L'extension `quarkus-reactive-messaging-smallrye-kafka`
- Configurer les Topics Kafka
- Configurer le déserialiseur et le serialiseur

# Connexion à 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);
    }

}
```

# Connexion à 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);
    }

}
```

# Connexion à 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);
    }

}
```

# Connexion à 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);
    }

}
```

# Connexion à 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);
    }

}
```

# Stratégies d'erreur

Faire en sorte que notre consommateur produise des erreurs occasionnelles:

```
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.

# Stratégies d'erreur

Faire en sorte que notre consommateur produise des erreurs occasionnelles:

```
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.

# Stratégies d'erreur

Faire en sorte que notre consommateur produise des erreurs occasionnelles:

```
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.

# Stratégies d'erreur

Nous pouvons configurer ce comportement avec:

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

Avec ce réglage, il logguera une erreur, mais continuera avec le flux.

# Stratégies d'erreur

Autre option:

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

Ce changement vers une *queue dead letter*, permet à un autre système - ou une opération humaine - de moniter ces messages, et éventuellement les reprogrammés.

# Stratégies de validation (Commit)

Kafka est un système de messagerie *at-least-once*. Pour indiquer à Kafka que vous avez traité avec succès un message, vous le validez (*commit*). Cela indiquera à Kafka que vous n'avez pas besoin de revoir ce message en particulier. Si vous ne validez pas de message, en cas de plantage et de redémarrage de votre application, vous verrez à nouveau le même message de Kafka.

SmallRye Reactive Messaging Kafka garde une trace de tous les accusés de réception des messages et, en fonction de cela, décide du moment de la validation.

# Stratégies de validation (Commit)

Supposons le connecteur passe les 8 message suivants à notre application et reçoit les accusés suivants:

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

Maintenant, il peut s'engager jusqu'au message n°3, car tout a été validé (commit) jusqu'au message numéro 3.

# Commit Strategies

Il y a trois stratégies de validation:

- Ignore
- Latest
- Throttled

# Exercise: Kafka

# Exercise: Dead Letter Queue & Stream filtering

# Fonctionnalités techniques

# Connaissance attendues

A l'issue de ce module vous devriez:

- Savoir comment ajouter des vérification de la santé de l'application
- Comprendre les différentes approches pour collecter des métriques
- Comprendre comment ajouter de la traçabilité

# Supervision

- L'action et les outils pour observer notre application sur une large période
  - Observer le statut global et mesurer le comportement
- C'est particulièrement pertinent dans un contexte d'environnements cloud (et kubernetes)

# Liveness and Readiness

- L'extension Smallrye Health de Quarkus implémente la spécification MicroProfile Health
- Ajouter cette extension fournit immédiatement des points d'entrées
  - /health
  - /health/live
  - /health/ready
- Liveness - qu'on peut traduire par vitalité - Le service est-il *démarré ou éteint, joignable ou injoignable* ?
- Readiness - qu'on peut traduire par le fait d'être prêt - Le service peut-il traiter des requêtes utilisateurs?
  - Un service peut être démarré et passer la vérification du Liveness mais échouer la vérification de Readiness
- Ceci correspond aux sondes liveness et readiness de Kubernetes

# Métriques (avec MicroProfile Metrics)

- L'extension Quarkus Smallrye Metrics implémente la spéc. MicroProfile Metrics
- Ajouter cette extension fournit immédiatement des points d'entrées:
  - /metrics
  - /metrics/base
  - /metrics/vendor
  - /metrics/application
- Supporte des retours en JSON et OpenMetrics
  - OpenMetrics est un projet de la sandbox CNCF (Cloud Native Computing Fundation)
  - Le standard OpenMetrics est basé sur les représentations de Prometheus

# Métriques (avec Micrometer)

- L'approche recommandée !
- Micrometer est le pendant de SLF4J pour les métriques
  - Une façade indépendante des implémentations éditeurs
- Extension Quarkus Micrometer avec une registry Prometheus
- Ajouter cette extension fournit immédiatement le point d'entrée `/metrics`
  - Au format Prometheus, avec des métriques applicatives comme système, JVM et HTTP
  - Le format JSON peut également être activé par la configuration

# Traces

- L'extension Quarkus Smallrye OpenTracing implémente la spec. MicroProfile OpenTracing
- Utilise le traçeur Jaeger
- Les identifiant de traces peuvent être enregistrés avec la propagation MDC
- Il est possible de configurer le suivi des requêtes JDBC et de la remise des messages Kafka

# Exercise: Observability (Bonus)

- Ajouter dans `application.properties`

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

- Ajouter un JBoss Logger dans `ProductsResource` et `PriceUpdatesResource` et logger quelques endpoints

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

- Ajouter les extensions:
  - Smallrye Health
  - Quarkus Micrometer
  - Quarkus Smallrye OpenTracing

# Recap

Dans ce module, nous avons vus:

- Comment ajouter des vérifications de liveness et readiness
- Deux façons permettant la collecte de métriques dans notre application Quarkus
- Comment activer le traçage distribué

# Conclusion

## Ressources

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

Merci