UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Réactive

La programmation réactive

1. **Modèle de programmation réactif** : Quarkus prend en charge la programmation réactive, ce qui permet de gérer efficacement les opérations asynchrones et les flux de données. Cela est particulièrement utile pour les applications nécessitant une haute performance et une faible latence.
2. **Intégration de Vert.x** : Quarkus s'intègre étroitement avec Vert.x, un toolkit pour la création d'applications réactives sur la JVM. Cela permet à Quarkus de gérer les requêtes de manière non-bloquante et d'optimiser les performances pour les opérations d'entrée/sortie.
3. **Extension Reactive Messaging** : Quarkus propose une extension pour le messaging réactif, permettant une communication asynchrone entre différents services et microservices. Cette extension supporte divers brokers de messages comme Kafka, AMQP, MQTT, etc.

La programmation réactive

4. **Extension Mutiny** : Mutiny est une bibliothèque réactive conçue pour une utilisation avec Quarkus. Elle offre une approche plus fluide et moins complexe pour la programmation réactive, avec des concepts comme `Uni` pour les opérations asynchrones à valeur unique, et `Multi` pour les flux de données.
5. **Support des bases de données réactives** : Quarkus permet de connecter des bases de données réactives, telles que PostgreSQL réactif, pour des opérations de base de données non-bloquantes.
6. **Configuration réactive** : Quarkus supporte également la configuration réactive, permettant aux applications de réagir dynamiquement aux changements de configuration.

La programmation réactive

1. **Création d'une Route Réactive avec Vert.x**

```java
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
public class ReactiveRoute {

    public void setupRouter(@Observes Router router) {
        router.route("/hello").handler(this::helloHandler);
    }

    private void helloHandler(RoutingContext rc) {
        rc.response().end("Hello from Reactive Route!");
    }
}
```

Ce code crée une route réactive qui répond à `/hello` en envoyant un simple message.

La programmation réactive

2. **Utilisation de Mutiny pour les Opérations Asynchrones**

```
import io.smallrye.mutiny.Uni;

public class ReactiveService {

    public Uni<String> getGreeting() {
        return Uni.createFrom().item(() -> "Hello, Reactive World!");
    }
}
```

Ici, `Uni` est utilisé pour encapsuler une opération asynchrone qui retourne un salut.

La programmation réactive

3. **Communication Réactive avec Kafka**

```java
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.eclipse.microprofile.reactive.messaging.Outgoing;

@ApplicationScoped
public class KafkaProcessor {

    @Incoming("source-topic")
    @Outgoing("sink-topic")
    public String process(String message) {
        return "Processed: " + message;
    }
}
```

Ce code démontre un simple traitement de message Kafka, où les messages sont lus d'un sujet source, traités, puis envoyés à un autre sujet.

La programmation réactive

4. Interagir avec une Base de Données Réactive

```
import io.smallrye.mutiny.Uni;
import io.vertx.mutiny.pgclient.PgPool;

@ApplicationScoped
public class DatabaseService {

    @Inject
    PgPool client;

    public Uni<String> findById(Long id) {
        return client.preparedQuery("SELECT name FROM table WHERE id = $1", Tuple.of(id))
                     .onItem().transform(pgRowSet -> {
                         if (pgRowSet.size() == 0) {
                             return "Not Found";
                         } else {
                             return pgRowSet.iterator().next().getString("name");
                         }
                     });
    }
}
```

Cet exemple montre comment effectuer une requête asynchrone à une base de données PostgreSQL réactive.

La programmation réactive Mutiny

`Uni` et `Multi` sont des concepts fondamentaux dans la bibliothèque Mutiny, qui est utilisée dans Quarkus pour la programmation réactive. Ils ne sont pas des threads en eux-mêmes, mais plutôt des abstractions pour gérer des opérations asynchrones de manière non-bloquante. Voici comment ils fonctionnent :

1. **Uni**

- **Usage**: `Uni` est utilisé pour représenter une opération asynchrone qui produit un seul résultat ou échoue. Il est similaire à `Future` mais offre plus de flexibilité et de contrôle.
- **Comportement**: Quand un `Uni` est créé, l'opération qu'il représente n'est pas encore exécutée. L'exécution commence seulement quand on s'abonne à ce `Uni`.
- **Exemple de Cas d'Usage**: Récupérer un résultat d'une base de données ou faire un appel HTTP.

2. **Multi**

- **Usage**: `Multi` représente un flux de données. Il peut émettre zéro, un ou plusieurs éléments, suivi d'une complétion ou d'une erreur.
- **Comportement**: Comme pour `Uni`, les données d'un `Multi` ne commencent à être émises qu'après une souscription. `Multi` peut être utilisé pour représenter des flux de données continus ou des événements.
- **Exemple de Cas d'Usage**: Traiter des flux de données en temps réel, comme des messages provenant d'un système de messagerie Kafka.

La programmation réactive Mutiny

3. **Fonctionnement Interne et Relation avec les Threads**

- **Asynchronie sans Blocage**: `Uni` et `Multi` permettent une programmation asynchrone sans bloquer les threads. Ils utilisent des callbacks et d'autres mécanismes non-bloquants pour traiter les résultats ou les flux de données.
- **Gestion des Threads**: Sous-jacent à ces abstractions, il y a une gestion intelligente des threads. Mutiny et Quarkus s'efforcent de minimiser l'utilisation de threads et de contextes de changement de thread, ce qui est crucial pour les performances et l'efficience, surtout dans des environnements à haute concurrence comme les microservices.
- **Intégration avec des Pools de Threads**: Dans les scénarios où les opérations bloquantes sont inévitables (par exemple, des appels de blocage IO), Mutiny et Quarkus peuvent déplacer ces opérations sur des pools de threads dédiés pour maintenir la non-blocage de la boucle d'événements principale.

La programmation réactive Mutiny

Fonctionnement de Uni

1. **Représentation d'une Opération Asynchrone**: `Uni` représente une opération asynchrone qui va produire un seul résultat à l'avenir. Ce résultat peut être une valeur réussie ou une erreur.
2. **Lazy Execution**: L'opération encapsulée par un `Uni` n'est pas exécutée immédiatement lors de sa création. Elle est exécutée seulement quand un abonné s'y inscrit, ce qui signifie que `Uni` est lazy.
3. **Abonnement et Callbacks**: Lorsqu'un consommateur s'abonne à un `Uni`, il fournit des callbacks pour gérer le résultat ou l'erreur. Le `Uni` invoque le callback approprié une fois que l'opération est terminée.
4. **Chainage des Opérations**: `Uni` permet de chaîner plusieurs opérations asynchrones de manière fluide en utilisant des méthodes comme `onItem()`. Cela permet de transformer les données ou d'effectuer d'autres opérations asynchrones une fois le résultat initial disponible.

La programmation réactive Mutiny

Fonctionnement de Multi

1. **Flux de Données**: `Multi` représente un flux de zéro, un ou plusieurs éléments. Ces éléments sont émis au fil du temps, ce qui le rend idéal pour représenter des flux de données ou des événements.
2. **Abonnement et Flux de Données**: Comme pour `Uni`, l'exécution d'un `Multi` commence lorsqu'un abonné s'y inscrit. L'abonné reçoit les éléments au fur et à mesure de leur émission.
3. **Opérations sur les Flux**: `Multi` offre des opérations pour manipuler le flux de données, telles que filtrer, mapper, réduire, et regrouper. Ces opérations permettent un traitement complexe et asynchrone des séquences de données.

La programmation réactive Mutiny

Gestion Sous-Jacente des Ressources

- **Événements Asynchrones**: `Uni` et `Multi` gèrent les événements de manière asynchrone, en utilisant des mécanismes internes pour écouter et réagir aux résultats des opérations asynchrones.
- **Utilisation Optimale des Threads**: Plutôt que de bloquer les threads en attendant les résultats, `Uni` et `Multi` utilisent des callbacks et des événements pour notifier les abonnés. Cela permet une utilisation efficace des ressources, en particulier dans des environnements parallèles et réactifs.
- **Intégration avec l'Écosystème Réactif**: Mutiny, et par extension `Uni` et `Multi`, s'intègrent bien avec d'autres outils et bibliothèques réactifs, permettant une programmation réactive cohérente et performante dans les applications Quarkus.

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Sécurité avec Quarkus

Sécurité avec Quarkus

La sécurité des microservices en Quarkus est un aspect crucial pour garantir que les applications sont résilientes, fiables et protégées contre les menaces potentielles. Quarkus, en tant que framework Java moderne, offre une multitude d'options pour sécuriser les applications microservices, en tirant parti de normes et de pratiques de l'industrie bien établies.

1. **Sécurisation des Points de Terminaison** : Les points de terminaison REST ou GraphQL sont souvent les premières lignes de défense. Quarkus fournit des moyens de sécuriser ces points de terminaison via l'authentification et l'autorisation, en utilisant des JWT, OAuth2, ou d'autres mécanismes standards.
2. **Isolation des Services** : Dans une architecture microservices, chaque service doit être isolé pour minimiser l'impact d'une faille de sécurité. Cela signifie déployer des services dans des conteneurs ou des environnements isolés et limiter leur communication à des interfaces bien définies.
3. **Communication Sécurisée** : Lorsque les services communiquent entre eux, il est crucial de sécuriser ces communications. SSL/TLS est fréquemment utilisé pour chiffrer le trafic réseau entre services.
4. **Gestion des Secrets** : Les applications Quarkus peuvent intégrer des systèmes de gestion des secrets pour stocker et accéder de manière sécurisée aux mots de passe, clés API, et autres données sensibles.
5. **Audit et Logging** : Garder des traces détaillées des activités et des événements de sécurité est essentiel pour surveiller l'intégrité du système et répondre rapidement aux incidents.

Sécurité avec Quarkus

Architecture Recommandée

1. **Architecture Orientée Services (SOA)** : Chaque microservice est développé, déployé et géré de manière indépendante. Cela permet de mettre à jour ou de remplacer des services individuels sans impacter l'ensemble du système.
2. **Conteneurisation et Orchestration** : Utiliser des conteneurs pour déployer des microservices permet une isolation, une scalabilité et une gestion efficace. Des outils comme Docker et Kubernetes sont souvent utilisés pour l'orchestration des conteneurs.
3. **API Gateway** : Un API Gateway agit comme un point d'entrée unique pour les clients externes, offrant une couche supplémentaire de sécurité, de gestion du trafic et de routage.
4. **Service Mesh** : Dans des architectures complexes, un service mesh comme Istio peut être utilisé pour gérer la communication sécurisée entre services, avec des fonctionnalités telles que le chiffrement de bout en bout et la gestion des politiques.
5. **Zero Trust Network** : Adopter une approche de sécurité "Zero Trust" où chaque demande de service est vérifiée, indépendamment de l'emplacement ou du réseau.
6. **Principes de Sécurité par la Conception** : Intégrer des considérations de sécurité dès les premières étapes de la conception et du développement.

Sécurité avec Quarkus JWT

JWT, qui signifie JSON Web Token, est une méthode standardisée pour la sécurisation des communications entre deux parties, souvent utilisée dans les applications web. Dans le contexte de Quarkus, un framework Java conçu pour les microservices et les applications cloud-native, JWT est utilisé pour gérer l'authentification et l'autorisation.

1. **Authentification** : Lorsqu'un utilisateur se connecte à une application Quarkus, ses informations d'identification sont vérifiées. Si elles sont valides, un JWT est créé et signé par le serveur. Ce token contient généralement des informations sur l'utilisateur, comme son identifiant et ses rôles.
2. **Transmission du Token** : Le JWT est envoyé à l'utilisateur, généralement dans un en-tête HTTP ou un cookie.
3. **Accès aux Ressources Protégées** : Lorsque l'utilisateur souhaite accéder à une ressource protégée, il envoie le JWT en tant que preuve de son identité et de ses autorisations.
4. **Validation du Token** : À chaque requête, le serveur Quarkus vérifie la validité du JWT (sa signature et sa date d'expiration, par exemple) avant de donner accès à la ressource demandée.
5. **Gestion des Rôles** : Quarkus peut utiliser les informations contenues dans le JWT pour autoriser l'accès à certaines parties de l'application basées sur les rôles de l'utilisateur.

Sécurité avec Quarkus JWT

1. Ajouter les Dépendances

Tout d'abord, assurez-vous d'inclure les dépendances nécessaires dans votre fichier `pom.xml` :

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-jwt</artifactId>
</dependency>
```

2. Configuration de JWT

Configurez JWT dans le fichier `application.properties` :

```
mp.jwt.verify.publickey.location=META-INF/resources/publicKey.pem
mp.jwt.verify.issuer=your-issuer
```

Vous devez générer une paire de clés publique/privée pour signer et vérifier les JWTs. La clé publique doit être placée dans le dossier `src/main/resources/META-INF/resources/`.

Sécurité avec Quarkus JWT

3. **Création d'un JWT**

Voici un exemple simplifié de création de JWT :

```
import io.smallrye.jwt.build.Jwt;

public class TokenUtils {

    public static String generateTokenString(String username, String[] roles) {
        return Jwt.issuer("your-issuer")
                  .upn(username)
                  .groups(new HashSet<>(Arrays.asList(roles)))
                  .signWithPrivateKey(privateKey) // privateKey est votre clé privée chargée
                  .compact();
    }
}
```

Sécurité avec Quarkus JWT

4. **Ressource REST Sécurisée**

Ensuite, créez une ressource REST sécurisée :

```
import javax.annotation.security.RolesAllowed;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/secure")
public class SecureResource {

    @GET
    @RolesAllowed({"User", "Admin"})
    public String secureMethod() {
        return "Access Granted";
    }
}
```

Dans cet exemple, seuls les utilisateurs ayant les rôles "User" ou "Admin" peuvent accéder à cette méthode.

Sécurité avec Quarkus JWT

5. **Authentification de l'Utilisateur**

Lorsque l'utilisateur se connecte, vous devez générer un JWT et le retourner :

```
@Path("/auth")
public class AuthenticationResource {

    @POST
    public Response authenticateUser(UserCredentials credentials) {
        // Vérifier les identifiants de l'utilisateur...
        String token = TokenUtils.generateTokenString(credentials.getUsername(), new String[]{"User"});
        return Response.ok().entity(token).build();
    }
}
```

Création d'une extension quarkus

Pour créer une extension Quarkus en détail, suivez ces étapes :

1. **Initialisation du Projet Maven** :
   - Créez un projet Maven multi-modules.
   - Nommez le projet et ajoutez deux sous-modules : `runtime` et `deployment`.
2. **Module Runtime** :
   - Ajoutez les dépendances nécessaires dans le fichier `pom.xml`, comme `quarkus-core` et les bibliothèques spécifiques à votre extension.
   - Développez une classe de configuration avec des annotations `@ConfigRoot` et `@ConfigItem`.
   - Créez un producteur CDI pour fournir une instance de votre objet principal, en utilisant des annotations comme `@Produces`.
   - Implémentez un enregistreur (`@Recorder`) pour capturer les appels d'invocation.
3. **Module de Déploiement** :
   - Assurez-vous que le module de déploiement dépend du module de runtime.
   - Implémentez des processeurs de Build Step (`@BuildStep`) pour générer du bytecode via des enregistreurs, traiter la configuration et préparer le code pour l'exécution.