UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Microservices Quarkus

Sommaire

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Principe des architecture microservices

Principe des architecture microservices

Agilité et monolithes

Définition d'un Monolithe

- Un monolithe est une application traditionnelle où tous les processus et fonctions sont intégrés dans une seule base de code et exécutés comme une unité unique.

Inconvénients des Monolithes

- **Difficulté à s'adapter aux changements** : Les mises à jour ou les modifications nécessitent souvent de redéployer l'application entière.
- **Limitation de la technologie** : Les monolithes limitent généralement l'équipe à une pile technologique unique.
- **Scalabilité verticale** : La scalabilité nécessite souvent du matériel plus puissant, plutôt que la réplication du service.
- **Développement lent** : Plus le codebase devient gros, plus il est difficile de comprendre, modifier et tester.

Principe des architecture microservices

Agilité et microservices

Définition des Microservices

- Les microservices sont une approche architecturale qui structure une application comme une collection de services faiblement couplés, qui implémentent des capacités d'affaires.

Avantages des Microservices pour l'Agilité

- **Déploiement indépendant** : Les services peuvent être déployés séparément, ce qui accélère les cycles de release.
- **Diversité technologique** : Chaque service peut utiliser la pile technologique la mieux adaptée à ses besoins.
- **Scalabilité horizontale** : Les services peuvent être facilement multipliés sur plusieurs machines et gérés dynamiquement.
- **Expérimentation et innovation** : Il est plus facile de tester de nouvelles idées et technologies sur des services individuels sans perturber l'ensemble du système.

Principe des architecture microservices

Agilité et microservices

Définitions :

- **Service** : Une unité fonctionnelle qui a une responsabilité unique et peut opérer de manière indépendante.
- **Couplage faible** : Les services sont développés et déployés sans dépendances étroites les uns envers les autres.
- **Cohésion élevée** : Chaque service gère un ensemble de fonctionnalités étroitement liées.

Principe des architecture microservices

Avantages et inconvénients

Avantages des Microservices

- **Flexibilité** : Facilité de mise à jour et de maintenance.
- **Résilience** : Un service en défaillance n'affecte pas tout le système.
- **Scalabilité** : Possibilité d'ajuster les ressources pour chaque service indépendamment.
- **Alignement organisationnel** : Les équipes peuvent être organisées autour de fonctionnalités métiers.

Inconvénients des Microservices

- **Complexité opérationnelle** : Gérer de nombreux services peut être complexe.
- **Défis de communication inter-services** : Le réseau devient une dépendance critique.
- **Consistance des données** : La gestion des données distribuées peut être délicate.
- **Coûts de développement initiaux** : La mise en place initiale d’une architecture microservices peut nécessiter plus d’efforts et de ressources.

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Quarkus

Quarkus

1. **HotSpot et GraalVM**

- GraalVM est une machine virtuelle polyglotte, c'est-à-dire qu'elle supporte l'exécution de code écrit dans plusieurs langages de programmation.
    1. **Moteur JIT (Just-In-Time) :** GraalVM intègre un compilateur JIT de haute performance qui convertit le bytecode Java en code machine au moment de l'exécution, optimisant ainsi la performance des applications Java.
    2. **Compilation AOT (Ahead-Of-Time) :** Avec l'outil `native-image`, GraalVM peut compiler des applications en exécutables natifs. Cela signifie que le bytecode Java est converti en code machine avant l'exécution, réduisant ainsi le temps de démarrage et la consommation de mémoire des applications.
    3. **Support Polyglotte :** GraalVM peut exécuter non seulement du code Java, mais aussi des langages interprétés comme JavaScript, Ruby, R, Python, et WebAssembly. Elle utilise le framework Truffle, qui est une API pour la construction d'interpréteurs de langages qui s'exécutent sur GraalVM.
    4. **Interopérabilité :** GraalVM permet l'interopérabilité entre les langages. Par exemple, un programme Java peut appeler du code écrit en JavaScript, Python, ou tout autre langage supporté.

Quarkus

1. **HotSpot et GraalVM**
    5. **Optimisations :** GraalVM applique des optimisations avancées, notamment l'élimination des allocations d'objets inutiles (escape analysis), la vectorisation et l'inlining agressif, ce qui peut considérablement améliorer les performances.
    6. **Outils :** GraalVM comprend des outils de développement et de profilage tels que Chrome Inspector et VisualVM, permettant aux développeurs de déboguer et de profiler des applications polyglottes.
    7. **Écosystème :** GraalVM peut être étendu avec des bibliothèques et des frameworks, et elle est compatible avec l'écosystème Java existant, ce qui permet aux développeurs d'utiliser leurs outils et bibliothèques préférés.
    8. **GraalVM Native Image :** L'outil Native Image permet de compiler des applications Java en images natives autonomes qui ne nécessitent pas une JVM. Ces images natives sont légères et offrent un temps de démarrage rapide, idéal pour les environnements cloud et les microservices.
    9. **Sécurité :** En tant que JVM, GraalVM fournit un modèle de sécurité robuste, bénéficiant de la gestion de la mémoire et des exceptions propre à la JVM.

Quarkus

- Quarkus est un framework Java conçu pour les applications cloud natives et les microservices. Il est optimisé pour les environnements de conteneurs et permet de créer des exécutables natifs via GraalVM.
- Le développement avec le framework Quarkus implique plusieurs concepts clés, des commandes spécifiques pour gérer le cycle de vie de l'application, ainsi que l'utilisation de standards et d'extensions pour créer des applications web et microservices performantes.

Concepts Clés de Quarkus

1. **Bootstrap rapide** : Quarkus minimise le temps de démarrage de l'application, ce qui est crucial pour les environnements de cloud et les architectures microservices.
2. **Faible empreinte mémoire** : Il réduit l'empreinte mémoire, permettant aux applications de consommer moins de ressources.
3. **Extension model** : Quarkus est extensible, permettant aux développeurs d'ajouter des fonctionnalités via des extensions Quarkus spécifiques.
4. **Build-time metadata processing** : Quarkus effectue une grande partie de son traitement pendant la phase de construction, ce qui réduit la surcharge au moment de l'exécution.
5. **Proactive and reactive models** : Il prend en charge à la fois les modèles de programmation réactifs et proactifs, permettant de gérer les opérations synchrones et asynchrones efficacement.

Quarkus

6. **Imperative and reactive programming** : Quarkus permet la coexistence de code impératif traditionnel et de code réactif au sein de la même application.
7. **Container First** : Conçu avec une première classe de support pour les conteneurs Docker et les orchestrations comme Kubernetes.
8. **Unification of imperative and reactive** : Il fournit une expérience cohérente pour le développement d'applications, qu'elles soient impératives ou réactives.

Commandes Quarkus

- Pour utiliser Quarkus, voici quelques commandes courantes :
- **Création d'un projet Quarkus** :

```
mvn io.quarkus:quarkus-maven-plugin:create
```

- **Lancement du mode de développement** (hot reload) :

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

- **Compilation en image native** (si GraalVM est configuré) :

```
./mvnw package -Pnative
```

- **Construction d'un conteneur Docker** :

```
./mvnw package -Dquarkus.container-image.build=true
```

Utilisation de JAX-RS avec Quarkus

- JAX-RS est une spécification Java pour les services RESTful. Quarkus utilise une implémentation de JAX-RS appelée RESTEasy pour définir les points de terminaison REST. Voici comment vous pourriez les utiliser dans Quarkus :
- **Créer un Resource** :

```
@Path("/hello")
public class GreetingResource {

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

- **Injection de dépendances** :

```
@Inject
GreetingService service;
```

Configuration Quarkus

- Quarkus utilise un système de configuration unifié qui est flexible et extensible. La configuration dans Quarkus est souvent définie dans le fichier `application.properties` qui se trouve dans le dossier `src/main/resources`. Vous pouvez également utiliser des variables d'environnement, des fichiers de configuration système, ou des secrets Kubernetes pour surcharger les configurations.
- Exemple de fichier `application.properties` :

```
quarkus.datasource.db-kind=h2
quarkus.datasource.jdbc.url=jdbc:h2:mem:test;DB_CLOSE_DELAY=-1
quarkus.datasource.jdbc.driver=org.h2.Driver
quarkus.hibernate-orm.database.generation=drop-and-create
```

Dans cet exemple, la configuration définit la connexion à une base de données H2 et la stratégie de génération de schéma pour Hibernate ORM.

Extensions Quarkus

- Les extensions Quarkus jouent un rôle fondamental dans le développement d'applications avec Quarkus. Elles servent à intégrer des fonctionnalités supplémentaires et à optimiser l'exécution des applications pour le mode JVM ainsi que le mode natif compilé avec GraalVM. Voici quelques détails sur leur fonctionnement et utilisation :
- **Rôle des Extensions Quarkus**
    1. **Intégration Transparente** : Elles permettent une intégration transparente de bibliothèques populaires et de standards de l'industrie, facilitant le développement d'applications.
    2. **Optimisation à la Compilation** : Elles aident à optimiser le code lors de la compilation, en éliminant tout ce qui est inutile pour réduire la taille et le temps de démarrage de l'application.
    3. **Configuration Facile** : Elles apportent des configurations par défaut qui peuvent être facilement personnalisées via le fichier `application.properties`.
    4. **Support Natif** : Elles offrent un support pour la compilation en natif, assurant que les bibliothèques intégrées fonctionnent bien avec GraalVM.
    5. **Fonctionnalités Déclaratives** : Elles permettent de déclarer des besoins en ressources, des services, des tâches planifiées et plus encore de manière déclarative.

Extensions Quarkus

- **Exemples d'Extensions Quarkus**
- **quarkus-resteasy** : pour développer des services web RESTful avec JAX-RS.
- **quarkus-hibernate-orm** : pour intégrer Hibernate ORM pour la persistance des données.
- **quarkus-smallrye-health** : pour ajouter des points de terminaison de contrôle de santé suivant la spécification MicroProfile Health.
- **quarkus-smallrye-metrics** : pour exposer des métriques d'application selon la spécification MicroProfile Metrics.
- **quarkus-smallrye-reactive-messaging** : pour la communication asynchrone entre services via des messages.
- **quarkus-vertx-web** : pour utiliser Vert.x Web pour créer des applications réactives.

Ajouter des Extensions à un Projet Quarkus

- Pour ajouter une extension à un projet Quarkus, utilisez la commande suivante :

```
./mvnw quarkus:add-extension -Dextensions="nom-de-l'extension"
```

- Par exemple, pour ajouter le support de PostgreSQL :

```
./mvnw quarkus:add-extension -Dextensions="quarkus-jdbc-postgresql"
```

Quarkus

Chargement à chaud (Hot Reload) :

- Quarkus propose un mode de développement avec rechargement à chaud. Lorsque vous démarrez votre application en mode développement en utilisant `mvn quarkus:dev`, Quarkus surveille les fichiers sources. Si vous apportez des modifications au code ou aux fichiers de configuration pendant que l'application est en cours d'exécution, Quarkus recompile et redémarre l'application automatiquement.
- Ce mode de développement est l'un des avantages de Quarkus, car il permet une boucle de feedback rapide, où les développeurs peuvent voir immédiatement l'effet de leurs modifications sans avoir besoin de redémarrer manuellement l'application.
    1. Démarrer l'application en mode développement : `./mvnw quarkus:dev`.
    2. Modifier le code source ou les fichiers de configuration.
    3. Quarkus détecte les changements et recompile les classes modifiées.
    4. L'application est automatiquement redéployée avec les modifications.

Ce processus simplifie grandement le développement en éliminant le besoin de cycles de build et de déploiement longs, permettant ainsi une itération rapide pendant le développement.

Utilisation des Profils Quarkus

- Les profils Quarkus sont des configurations spécifiques à l'environnement qui vous permettent de personnaliser le comportement de votre application pour différents environnements, comme le développement, les tests ou la production. Ils sont très utiles pour gérer les paramètres qui varient selon l'environnement sans avoir à modifier le code ou recourir à des techniques de configuration plus complexes.
- Chaque profil est défini par un préfixe qui est utilisé dans le fichier `application.properties` ou tout autre fichier de configuration compatible. Par exemple, `%dev.quarkus.datasource.username` définit le nom d'utilisateur de la source de données pour le profil de développement.

```
# Configuration par défaut, utilisée si aucun profil n'est activé
quarkus.http.port=8080

# Configuration pour le profil 'dev'
%dev.quarkus.http.port=8081

# Configuration pour le profil 'test'
%test.quarkus.http.port=8082

# Configuration pour le profil 'prod'
%prod.quarkus.http.port=80
```

Utilisation des Profils Quarkus

- Quarkus définit trois profils par défaut :
  - `dev` : Utilisé lors de l'exécution en mode développement (`./mvnw quarkus:dev`).
  - `test` : Utilisé lors de l'exécution des tests.
  - `prod` : Le profil par défaut lorsque l'application est exécutée en mode production.

Définir et Activer les Profils

- Pour définir un profil personnalisé, vous ajoutez simplement le préfixe `%[nom-du-profil]` devant les propriétés de configuration que vous souhaitez surcharger pour ce profil.
- Pour activer un profil spécifique, vous pouvez utiliser la propriété système `quarkus.profile` lors du démarrage de votre application :

```
java -Dquarkus.profile=prod -jar mon-app.jar
```

Ou pour Maven, vous pouvez le passer comme argument :

```
./mvnw quarkus:dev -Dquarkus.profile=dev
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Patterns, vocabulaire et concepts liés aux microservices

Patterns, vocabulaire et concepts liés aux microservices

1. **Vocabulaire et Concepts Clés** :
   - **Conteneurisation** : L'encapsulation de services dans des conteneurs pour faciliter le déploiement et l'isolation.
   - **Orchestration** : La gestion des conteneurs et des services, souvent réalisée avec des outils comme Kubernetes.
   - **API Gateway** : Un point d'entrée pour les clients qui permet d'unifier et de gérer les requêtes vers différents services.
   - **Service Discovery** : Le processus permettant aux services de se découvrir et de communiquer entre eux dynamiquement.
   - **Circuit Breaker** : Un pattern qui permet de prévenir les défaillances en cascade.
   - **Load Balancing** : La distribution du trafic réseau ou des demandes sur plusieurs serveurs.
   - **Database per Service** : Chaque microservice gère sa propre base de données.
   - **CQRS (Command Query Responsibility Segregation)** : Séparation des opérations de lecture et d'écriture pour un système.
   - **Event Sourcing** : Persistance des changements en tant qu'une séquence d'événements.

Patterns, vocabulaire et concepts liés aux microservices

2. **Organisation et Structure** :
   - Découpage en services basé sur les domaines d'affaires (Domain-Driven Design).
   - Gestion des équipes organisées autour des fonctions des microservices (équipes cross-fonctionnelles).
3. **Patterns de Microservices** :
   - **Décomposition** : Diviser une application en services plus petits.
   - **Aggregator** : Un service qui agrège les résultats de plusieurs services.
   - **Proxy** : Un service agissant comme intermédiaire pour redistribuer ou filtrer les requêtes.
   - **Chained or Chain of Responsibility** : Un pattern où une requête passe à travers une chaîne de services.
   - **Branch** : Des flux de traitement parallèles qui se rejoignent ou se séparent.
4. **Sécurité dans les Microservices** :
   - Authentification et autorisation.
   - Patterns de sécurité comme OAuth2 et JWT.
5. **Observabilité et Monitoring** :
   - Importance de la surveillance en temps réel.
   - Utilisation de Prometheus, Grafana, et d'autres outils de monitoring.

Patterns, vocabulaire et concepts liés aux microservices

DDD

- Le Domain-Driven Design est une approche de développement logiciel axée sur la création de logiciels qui reflètent les complexités et les exigences intrinsèques du domaine d'activité auquel ils s'appliquent. L'objectif est de faciliter la conception de logiciels en modélisant le domaine d'affaires de manière profonde et significative.

1. **Modèle du Domaine**:
    - C'est une représentation structurée du domaine d'affaires, comprenant des entités, des objets de valeur, des services, des événements et des agrégats, qui définissent comment les éléments interagissent entre eux.
2. **Entité (Entity)**:
    - Une entité est un objet qui est défini par son identité, plutôt que par ses attributs. Cette identité persiste à travers le temps et les différentes représentations de l'entité.
3. **Objet de Valeur (Value Object)**:
    - À l'opposé de l'entité, un objet de valeur est défini par ses attributs et n'a pas d'identité distincte. Les objets de valeur sont immuables et sont souvent utilisés pour mesurer, quantifier ou décrire quelque chose dans le domaine.

Patterns, vocabulaire et concepts liés aux microservices

DDD

4. **Agrégat**:
    - Un agrégat est une collection d'objets (entités et objets de valeur) qui sont traités comme une unité cohérente pour les transactions de données. Chaque agrégat a une racine et une limite bien définie autour de ses composants.
5. **Racine d'Agrégat (Aggregate Root)**:
    - C'est l'entité principale au sein d'un agrégat, à travers laquelle toutes les interactions extérieures avec l'agrégat doivent passer. Elle garantit la cohérence de l'agrégat en contrôlant l'accès et en appliquant les règles d'intégrité.
6. **Contexte Délimité (Bounded Context)**:
    - Il s'agit d'une limite conceptuelle au sein de laquelle un modèle de domaine particulier est défini et applicable. Il permet de séparer et de maintenir l'intégrité des différents modèles du domaine dans un système complexe.

Patterns, vocabulaire et concepts liés aux microservices

Application du DDD dans le Découpage en Microservices:

- Imaginons une application de commerce électronique avec plusieurs microservices :

1. **Microservice Client** :
   - **Entité** : Client (identifié par un ID client)
   - **Objets de Valeur** : Email, Adresse
   - **Agrégat** : Compte Client avec ses informations personnelles et ses préférences
2. **Microservice Produit** :
   - **Entité** : Produit (identifié par un SKU ou ID de produit)
   - **Objets de Valeur** : Description du produit, Spécifications
   - **Agrégat** : Informations sur le produit incluant le prix, le stock, et les fournisseurs
3. **Microservice Commande** :
   - **Entité** : Commande (identifiée par un numéro de commande)
   - **Objets de Valeur** : Ligne de commande (quantité, prix unitaire)
   - **Agrégat** : Commande comprenant les lignes de commande, les informations d'expédition, et le statut

Patterns, vocabulaire et concepts liés aux microservices

Exercice : Conception d'une Application de Service de Covoiturage avec DDD

Contexte : votre mission est de concevoir une application de service de covoiturage qui met en relation les conducteurs et les passagers. L'application doit permettre aux utilisateurs de réserver un trajet, aux conducteurs de publier leurs itinéraires disponibles, et doit inclure un système de paiement et d'évaluation.

Objectif :

Utiliser le Domain-Driven Design pour identifier les entités, les objets de valeur, les agrégats et les contextes délimités pour structurer l'application en microservices.

Instructions :

1. **Identification des Fonctionnalités :**
   - Listez les fonctionnalités principales de l'application, telles que la recherche de trajet, la publication d'itinéraire, la réservation, le paiement, et les évaluations.
2. **Modélisation du Domaine :**
   - Déterminez les entités principales comme Trajet, Utilisateur (Conducteur, Passager), Véhicule, Paiement, Évaluation.
   - Identifiez les objets de valeur comme Position GPS, Tarification, Commentaires.

Patterns, vocabulaire et concepts liés aux microservices

3. **Définition des Agrégats :**
    - Formez des agrégats autour des entités principales. Par exemple, un agrégat Trajet pourrait inclure Conducteur, Itinéraire, Passager, et Paiement.
4. **Délimitation des Contextes :**
    - Divisez le modèle en contextes délimités en fonction des sous-systèmes fonctionnels tels que "Gestion des Utilisateurs", "Planification des Trajets", "Gestion des Paiements", et "Système d'Évaluations".