

9. Web Services

9.1 Activité pratique : JAX-RS / Jersey

1. Création du projet avec Spring Initializr

Il est nécessaire de créer un projet à l'aide de Spring Initializr (<https://start.spring.io/>). Les dépendances suivantes sont requises :

- Spring Web
- Spring Data JPA
- H2 Database (pour utiliser une base de données en mémoire)
- Jersey (pour l'intégration de JAX-RS)
- Lombok (pour générer automatiquement les getters, setters, etc.)
- DevTools.

Définir le groupe et l'artifact pour le projet. Après cela, télécharger le projet généré sous forme d'archive ZIP, l'extraire, puis l'ouvrir dans un IDE comme IntelliJ IDEA ou Eclipse.

2. Configuration de la base de données H2

Dans le fichier application.properties, configurer la source de données H2 ainsi que le port du serveur. Exemple de configuration :

```
spring.datasource.url=jdbc:h2:mem:banque
server.port=8082
```

Une base de données H2 en mémoire est utilisée ici, nommée banque. L'application sera accessible sur le port 8082.

3. Définition de l'entité JPA

Créer une classe Compte représentant l'entité dans la base de données, annotée avec @Entity. Utiliser @Id pour spécifier la clé primaire et @GeneratedValue pour que l'ID soit généré automatiquement.

Exemple :

```

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Compte {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double solde;
    @Temporal(TemporalType.DATE)
    private Date dateCreation;
    @Enumerated(EnumType.ORDINAL)
    private TypeCompte type;
}

```

Cette entité Compte comprend un identifiant, un solde, une date de création et un type de compte (courant ou épargne). Le type de compte est défini par une énumération TypeCompte, comme illustré ci-dessous.

4. Définition de l'énumération TypeCompte

L'énumération TypeCompte permet de définir les types de comptes disponibles, à savoir les comptes courants (COURANT) et les comptes épargne (EPARGNE).

Exemple :

```

public enum TypeCompte {
    COURANT, EPARGNE
}

```

Cette énumération est utilisée dans la classe Compte pour définir la nature du compte créé.

5. Création du dépôt JPA (Repository)

Créer une interface CompteRepository qui étend JpaRepository. Cela permet d'utiliser directement des méthodes prédéfinies telles que findAll(), save(), deleteById(), etc.

Exemple :

```

public interface CompteRepository extends JpaRepository<Compte, Long> {
}

```

Cette interface permet de manipuler les données dans la base de données H2 sans avoir à implémenter manuellement les méthodes de base.

6. Initialisation des données

Avant de configurer l'API, il est nécessaire d'initialiser quelques données dans la base de données H2 en utilisant un CommandLineRunner. Cela permet de pré-remplir des comptes dans la base de données au démarrage de l'application.

Exemple de méthode d'initialisation :

```

@SpringBootApplication
public class MsBanqueApplication {
    public static void main(String[] args) {
        SpringApplication.run(MsBanqueApplication.class, args);
    }

    @Bean
    CommandLineRunner start(CompteRepository compteRepository) {

```

```

        return args -> {
            compteRepository.save(new Compte(null, Math.random()*9000, new
                Date(), TypeCompte.EPARGNE));
            compteRepository.save(new Compte(null, Math.random()*9000, new
                Date(), TypeCompte.COURANT));
            compteRepository.save(new Compte(null, Math.random()*9000, new
                Date(), TypeCompte.EPARGNE));

            compteRepository.findAll().forEach(c -> {
                System.out.println(c.toString());
            });
        };
    }
}

```

Cette méthode initialise trois comptes aléatoires dans la base de données au démarrage de l'application. Les comptes créés sont de types EPARGNE et COURANT. Ils sont ensuite affichés dans la console pour vérification.

7. Configuration du Service REST avec Jersey

Dans la classe de configuration, il est nécessaire de définir un bean de type ResourceConfig pour enregistrer la classe RESTful JAX-RS. Cette classe contiendra les méthodes pour traiter les requêtes HTTP.

Exemple :

```

@Configuration
public class MyConfig {
    @Bean
    public ResourceConfig resourceConfig() {
        ResourceConfig jerseyServlet = new ResourceConfig();
        jerseyServlet.register(CompteRestJaxRSAPI.class);
        return jerseyServlet;
    }
}

```

Cette configuration indique à Spring d'utiliser Jersey pour gérer les requêtes HTTP. La classe CompteRestJaxRSAPI est enregistrée ici pour recevoir et traiter les requêtes.

8. Implémentation des services REST

Créer une classe CompteRestJaxRSAPI avec des annotations JAX-RS telles que @GET, @POST, @PUT, et @DELETE pour implémenter les opérations CRUD.

Exemple :

```

@Component
@Path("/banque")
public class CompteRestJaxRSAPI {

    @Autowired
    private CompteRepository compteRepository;

    // READ: Récupérer tous les comptes
    @Path("/comptes")
    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public List<Compte> getComptes() {

```

```

        return compteRepository.findAll();
    }

    // READ: Récupérer un compte par son identifiant
    @Path("/comptes/{id}")
    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public Compte getCompte(@PathParam("id") Long id) {
        return compteRepository.findById(id).orElse(null);
    }

    // CREATE: Ajouter un nouveau compte
    @Path("/comptes")
    @POST
    @Consumes({MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_JSON})
    public Compte addCompte(Compte compte) {
        return compteRepository.save(compte);
    }

    // UPDATE: Mettre à jour un compte existant
    @Path("/comptes/{id}")
    @PUT
    @Consumes({MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_JSON})
    public Compte updateCompte(@PathParam("id") Long id, Compte compte) {
        Compte existingCompte = compteRepository.findById(id).orElse(null);
        if (existingCompte != null) {
            existingCompte.setSolde(compte.getSolde());
            existingCompte.setDateCreation(compte.getDateCreation());
            existingCompte.setType(compte.getType());
            return compteRepository.save(existingCompte);
        }
        return null;
    }

    // DELETE: Supprimer un compte
    @Path("/comptes/{id}")
    @DELETE
    @Produces({MediaType.APPLICATION_JSON})
    public void deleteCompte(@PathParam("id") Long id) {
        compteRepository.deleteById(id);
    }
}

```

Ce service REST expose plusieurs endpoints pour gérer les opérations CRUD (Create, Read, Update, Delete) sur les comptes bancaires. Les endpoints permettent de récupérer, ajouter, mettre à jour et supprimer des comptes dans la base de données H2. Voici une description des principaux endpoints :

- GET /banque/comptes : Ce point d'accès permet de récupérer l'ensemble des comptes disponibles dans la base de données. La méthode `getComptes()` est utilisée pour interroger la base de données via le dépôt `CompteRepository`. La réponse est renvoyée au format JSON. Ce type de requête est généralement utilisé pour afficher la liste complète des comptes à des fins de consultation ou de traitement analytique.
- GET /banque/comptes/{id} : Ce point d'accès permet de récupérer un compte spécifique en fonction de son id. L'identifiant est passé directement dans l'URL grâce à l'annotation

`@PathParam`. La méthode `getCompte(Long id)` interroge la base de données pour renvoyer le compte correspondant à cet identifiant, s'il existe. Sinon, la réponse peut être `null` ou un code d'erreur HTTP (tel que 404) indiquant que le compte n'a pas été trouvé.

Ces deux endpoints de type GET jouent un rôle clé dans la consultation des données, avec un focus sur deux scénarios :

- La récupération de l'ensemble des comptes : Utile pour l'affichage de la liste des comptes, des tableaux de bord ou la génération de rapports sur l'état des comptes (ex. solde total des comptes d'épargne ou des comptes courants).
- La récupération d'un compte spécifique : Utilisée dans les interfaces utilisateurs pour afficher les détails d'un compte particulier lors de la consultation ou de la modification de ses informations. Cette fonctionnalité est particulièrement utile dans le cadre d'opérations ciblées sur un client donné.

La méthode `getComptes()` renvoie une liste de comptes obtenue à partir de la base de données via `findAll()`. Les données sont formatées en JSON, permettant une intégration fluide avec des applications front-end. Quant à la méthode `getCompte(Long id)`, elle récupère un compte précis grâce à `findById()` et retourne les détails de ce compte en format JSON. Si le compte n'existe pas, une réponse vide ou un code HTTP approprié peut être renvoyé pour gérer le cas où le compte est introuvable.

9. Lancement de l'application

Pour lancer l'application, il suffit d'exécuter la classe `MsBanqueApplication` qui contient la méthode principale. Cette classe est annotée avec `@SpringBootApplication`, ce qui active la configuration automatique de Spring Boot.

Exemple :

```
@SpringBootApplication
public class MsBanqueApplication {
    public static void main(String[] args) {
        SpringApplication.run(MsBanqueApplication.class, args);
    }
}
```

L'application démarre sur le port configuré (8082 dans cet exemple).

10. Configuration pour supporter JSON et XML

Pour permettre au service REST de gérer à la fois les formats JSON et XML, il est nécessaire d'indiquer que les méthodes peuvent produire et consommer ces deux types de données. En JAX-RS, cela se fait à l'aide des annotations `@Produces` et `@Consumes`.

Voici les étapes pour configurer le service REST afin de prendre en charge à la fois JSON et XML :

10.1 Ajout de la dépendance JAXB pour XML

Spring Boot gère déjà Jackson par défaut pour le traitement JSON. Cependant, pour gérer les données au format XML, il est nécessaire d'utiliser JAXB. Si JAXB n'est pas inclus, il est possible d'ajouter la dépendance suivante dans le fichier `pom.xml` :

```
<dependency>
<groupId>javax.xml.bind</groupId>
<artifactId>jaxb-api</artifactId>
<version>2.3.1</version>
</dependency>
```

Cette dépendance permet d'ajouter le support XML à l'application. Il est important de vérifier que le projet est bien configuré pour intégrer les deux formats de sérialisation.

10.2 Modification des méthodes REST pour supporter JSON et XML

Voici une version mise à jour de la classe CompteRestJaxRSAPI, où chaque méthode peut maintenant traiter les formats JSON et XML. L'annotation @Produces spécifie les types de données que la méthode peut renvoyer, tandis que l'annotation @Consumes précise les formats de données que la méthode peut accepter.

```

@Component
@Path("/banque")
public class CompteRestJaxRSAPI {

    @Autowired
    private CompteRepository compteRepository;

    // READ: Récupérer tous les comptes (JSON et XML)
    @Path("/comptes")
    @GET
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public List<Compte> getComptes() {
        return compteRepository.findAll();
    }

    // READ: Récupérer un compte par son identifiant (JSON et XML)
    @Path("/comptes/{id}")
    @GET
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public Compte getCompte(@PathParam("id") Long id) {
        return compteRepository.findById(id).orElse(null);
    }

    // CREATE: Ajouter un nouveau compte (JSON et XML)
    @Path("/comptes")
    @POST
    @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public Compte addCompte(Compte compte) {
        return compteRepository.save(compte);
    }

    // UPDATE: Mettre à jour un compte existant (JSON et XML)
    @Path("/comptes/{id}")
    @PUT
    @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public Compte updateCompte(@PathParam("id") Long id, Compte compte) {
        Compte existingCompte = compteRepository.findById(id).orElse(null);
        if (existingCompte != null) {
            existingCompte.setSolde(compte.getSolde());
            existingCompte.setDateCreation(compte.getDateCreation());
            existingCompte.setType(compte.getType());
            return compteRepository.save(existingCompte);
        }
        return null;
    }
}

```

```
// DELETE: Supprimer un compte (JSON et XML)
@Path("/comptes/{id}")
@DELETE
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public void deleteCompte(@PathParam("id") Long id) {
    compteRepository.deleteById(id);
}
```

Explications :

- Chaque méthode est annotée avec `@Produces` pour indiquer que la méthode peut renvoyer soit du JSON, soit du XML en fonction de la requête du client.
- De même, les méthodes POST et PUT sont annotées avec `@Consumes` pour accepter des données au format JSON ou XML dans le corps de la requête.
- Si un client fait une requête avec un `Accept` header en `application/xml`, le service renverra une réponse en XML. Si le client préfère `application/json`, le service fournira une réponse en JSON.

10.3 Ajout de l'annotation `@XmlRootElement` pour le support XML

Pour permettre la sérialisation en XML, la classe Compte doit être annotée avec `@XmlRootElement`. Cette annotation indique que cette classe peut être convertie en XML et qu'elle représente la racine du document XML.

Voici la version modifiée de la classe Compte :

```
import javax.xml.bind.annotation.XmlRootElement;
import javax.persistence.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@XmlRootElement // Indique que cette classe peut être sérialisée en XML
public class Compte {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double solde;

    @Temporal(TemporalType.DATE)
    private Date dateCreation;

    @Enumerated(EnumType.ORDINAL)
    private TypeCompte type;
}
```

Explications :

- L'annotation `@XmlRootElement` permet à la classe Compte d'être considérée comme un élément racine d'un document XML lors de la sérialisation.
- Cette annotation est nécessaire pour que JAXB puisse convertir l'objet Compte en XML. Si un client demande une réponse en XML, le service REST pourra désormais renvoyer une représentation XML du compte.

10.4 Exemple de réponse XML

Lorsqu'un client demande une réponse au format XML en utilisant Accept: application/xml, le serveur retournera la structure XML suivante pour un objet Compte :

```
<compte>
    <id>1</id>
    <solde>5000.0</solde>
    <dateCreation>2024-10-30T00:00:00</dateCreation>
    <type>EPARGNE</type>
</compte>
```

Ce document XML représente les détails d'un compte, avec l'élément racine compte, défini par l'annotation @XmlRootElement.

10.5 Tests avec Postman ou Curl

Pour tester cette fonctionnalité, il est possible d'utiliser un outil comme Postman ou Curl en modifiant le header Accept pour demander soit du JSON, soit du XML.

Exemple d'une requête Curl pour obtenir la liste des comptes en JSON :

```
curl -X GET "http://localhost:8082/banque/comptes" -H "Accept: application/json"
```

Exemple d'une requête Curl pour obtenir la liste des comptes en XML :

```
curl -X GET "http://localhost:8082/banque/comptes" -H "Accept: application/xml"
```

Ces requêtes permettent de tester si le service retourne correctement le format demandé par le client.