

## 9.11 Activité pratique : Spring Data REST

### 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 Data REST
- Spring Data JPA
- H2 Database (pour utiliser une base de données en mémoire)
- 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.



Spring Data REST simplifie l'exposition des repositories en tant que services RESTful. Cette approche réduit considérablement le besoin de créer manuellement des contrôleurs pour les opérations CRUD de base.

### 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 :

---

```
# Configuration de la source de données H2
spring.datasource.url=jdbc:h2:mem:banque
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Activer la console H2
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# Configuration du serveur
server.port=8082

# Configuration Hibernate pour la création de la base de données
spring.jpa.hibernate.ddl-auto=update

# Définir le chemin de base pour les APIs Spring Data REST
spring.data.rest.base-path=/api
```

---

Une base de données H2 en mémoire est utilisée ici, nommée banque. L'application sera accessible sur le port 8082. De plus, la console H2 est activée et accessible via /h2-console pour faciliter la gestion de la base de données pendant le développement.



L'option `spring.jpa.hibernate.ddl-auto=update` permet à Hibernate de gérer automatiquement la création et la mise à jour des schémas de la base de données en fonction des entités définies.

### 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 :

---

```

import lombok.AllArgsConstructorConstructor;
import lombok.Data;
import lombok.NoArgsConstructorConstructor;

import javax.persistence.*;
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlElementWrapper;
import java.util.Date;

@Entity
@Data
@NoArgsConstructorConstructor
@AllArgsConstructorConstructor
public class Compte {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double solde;

    @Temporal(TemporalType.DATE)
    private Date dateCreation;

    @Enumerated(EnumType.STRING)
    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.



L'utilisation de `EnumType.STRING` dans l'annotation `@Enumerated` permet de stocker les valeurs de l'énumération sous forme de chaînes de caractères dans la base de données, améliorant ainsi la lisibilité des données stockées.

#### 5. Création du dépôt JPA (Repository)

Créer une interface CompteRepository qui étend JpaRepository. Spring Data REST exposera automatiquement ce repository en tant que service RESTful.

Exemple :

---

```

import ma.rest.spring.entities.Compte;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;
import org.springframework.stereotype.Repository;

```

---

---

```
@RepositoryRestResource
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. L'annotation `@RepositoryRestResource` permet de personnaliser le chemin d'accès des endpoints REST.



Spring Data REST expose automatiquement les endpoints RESTful basés sur les méthodes définies dans `JpaRepository`. Par exemple, des endpoints pour les opérations CRUD sont générés sans nécessiter de contrôleur supplémentaire.

## 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 :

---

```
import ma.rest.spring.entities.Compte;
import ma.rest.spring.entities.TypeCompte;
import import ma.rest.spring.repositories.CompteRepository;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

import java.util.Date;

@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.



L'utilisation de `CommandLineRunner` permet d'exécuter du code au démarrage de l'application, facilitant ainsi l'initialisation des données de test sans nécessiter d'interaction manuelle.

## 7. Configuration du Service REST avec Spring Data REST

Spring Data REST permet d'exposer automatiquement les repositories en tant que services RESTful sans nécessiter la création de contrôleurs manuels. Cette section détaille la configuration et la personnalisation du service REST.

### a. Exposition Automatique des Repositories

Avec Spring Data REST, il n'est pas nécessaire de créer des contrôleurs REST pour les opérations CRUD de base. En étendant `JpaRepository` et en annotant le repository avec `@RepositoryRestResource`, Spring Data REST expose automatiquement les endpoints REST correspondants.

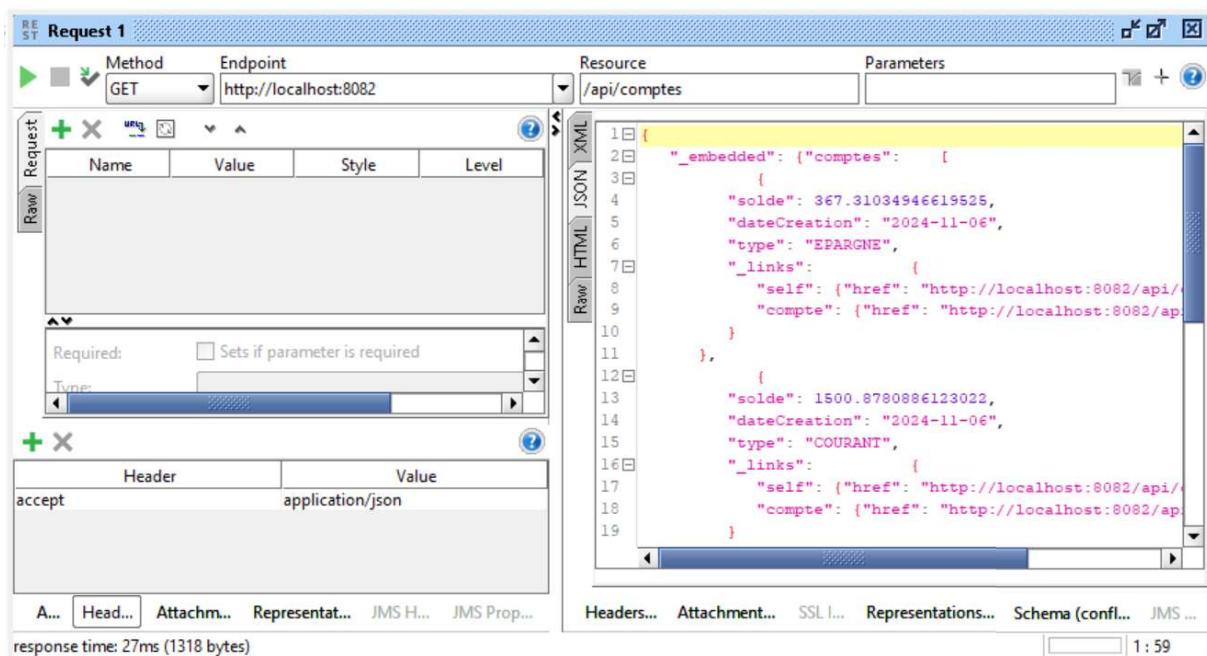


Figure 9.10: Test avec SoapUI.



Les endpoints générés incluent des opérations telles que `GET /comptes`, `POST /comptes`, `PUT /comptes/id`, et `DELETE /comptes/id`. Ces endpoints facilitent l'interaction avec les entités sans nécessiter de code supplémentaire.

### b. Personnalisation des Endpoints REST

L'annotation `@RepositoryRestResource` permet de personnaliser le chemin d'accès des endpoints REST.

Exemple :

---

```

@RepositoryRestResource(path = "comptes", collectionResourceRel = "comptes",
    itemResourceRel = "compte")
public interface CompteRepository extends JpaRepository<Compte, Long> {
}
  
```

---

Dans cet exemple :

- path = "comptes" : Définit le chemin de base pour les endpoints REST (par exemple, /comptes).
- collectionResourceRel = "comptes" : Définit le nom de la collection dans les réponses.
- itemResourceRel = "compte" : Définit le nom de l'élément individuel dans les réponses.



La personnalisation de la gestion des exceptions permet d'améliorer la clarté des messages d'erreur retournés aux clients et d'assurer une meilleure expérience utilisateur en cas d'erreurs.

## 8. Gestion des Projections avec Spring Data REST

Spring Data REST permet de créer des projections pour contrôler les informations retournées par les endpoints REST. Dans cette activité, nous allons créer deux projections : une projection pour afficher uniquement le solde du compte (CompteProjection1) et une autre projection pour afficher le solde et le type de compte (CompteProjection2).

### a. Crédation des Projections

---

```
import org.springframework.data.rest.core.config.Projection;

@Projection(name = "solde", types = Compte.class)
public interface CompteProjection1 {
    public double getSolde();
}

@Projection(name = "mobile", types = Compte.class)
public interface CompteProjection2 {
    public double getSolde();
    public TypeCompte getType();
}
```

---



Les projections permettent de simplifier les réponses des APIs en limitant les informations retournées. La projection CompteProjection1 renvoie uniquement le solde, tandis que CompteProjection2 inclut également le type de compte.

### b. Utilisation des Projections dans les Requêtes

Pour utiliser une projection, il suffit d'ajouter le paramètre projection dans l'URL. Par exemple :

- `http://localhost:8082/api/comptes/1?projection=solde` : Affiche uniquement le solde du compte.
- `http://localhost:8082/api/comptes/1?projection=mobile` : Affiche le solde et le type du compte.

## 9. Exposition des IDs dans les Réponses JSON

Pour exposer les IDs dans les réponses JSON, il est nécessaire de configurer RepositoryRestConfiguration. Exemple :

---

```
import org.springframework.data.rest.core.config.RepositoryRestConfiguration;

@Bean
CommandLineRunner start(CompteRepository compteRepository,
    RepositoryRestConfiguration restConfiguration) {
    return args -> {
        restConfiguration.exposeIdsFor(Compte.class);
        // Initialisation des comptes
    }
}
```

---

```

        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));
    };
}

```

---



La méthode `exposeIdsFor` permet d'inclure les IDs des entités dans les réponses JSON, ce qui facilite leur identification et manipulation côté client.

## 10. Configuration de la Pagination et du Tri dans les Requêtes

Pour gérer de grandes quantités de données, Spring Data REST permet d'ajouter des paramètres de pagination et de tri dans les requêtes. Exemple :

- `http://localhost:8082/api/comptes?page=0&size=2` : Affiche les comptes avec une pagination de 2 comptes par page.
- `http://localhost:8082/api/comptes?page=0&size=2&sort=solde,desc` : Trie les comptes par solde en ordre décroissant.



La pagination et le tri améliorent les performances de l'application en réduisant la charge de données lors des requêtes et en permettant aux utilisateurs de naviguer efficacement dans les résultats.

## 11. Gestion des Liens entre les Ressources Client et Compte

Dans une application bancaire, chaque client peut avoir plusieurs comptes. Spring Data REST permet de définir les relations entre les entités, ce qui facilite l'exposition des liens entre les ressources. Nous allons créer une relation entre Client et Compte pour afficher les comptes associés à chaque client.

### a. Définition de l'Entité Client

Commencez par créer une entité Client avec une relation `@OneToMany` vers Compte.

```

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;
import java.util.List;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Client {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String email;

    @OneToMany(mappedBy = "client")
    private List<Compte> comptes;
}

```

---



L'annotation `@OneToMany` crée une relation entre `Client` et `Compte`, où un client peut avoir plusieurs comptes. L'attribut `mappedBy` indique que cette relation est contrôlée par l'entité `Compte`.

### b. Mise à jour de l'Entité Compte

Modifiez l'entité `Compte` pour ajouter une relation `@ManyToOne` vers `Client`.

---

```
import javax.persistence.*;

@Entity
public class Compte {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double solde;

    @Temporal(TemporalType.DATE)
    private Date dateCreation;

    @Enumerated(EnumType.STRING)
    private TypeCompte type;

    @ManyToOne
    @JoinColumn(name = "client_id")
    private Client client;
}
```

---



L'annotation `@ManyToOne` indique que chaque compte est associé à un seul client. L'annotation `@JoinColumn` permet de définir le nom de la colonne de clé étrangère dans la table des comptes.

### c. Initialisation des Données Client et Compte

Pour initialiser des données pour les clients et leurs comptes associés dans la base de données H2, nous allons utiliser un `CommandLineRunner`. Cela nous permettra de créer deux clients et d'associer des comptes à chacun d'eux au démarrage de l'application.

---

```
import ma.rest.spring.entities.Compte;
import ma.rest.spring.entities.Client;
import ma.rest.spring.entities.TypeCompte;
import ma.rest.spring.repositories.CompteRepository;
import ma.rest.spring.repositories.ClientRepository;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.data.rest.core.config.RepositoryRestConfiguration;

import java.util.Date;

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

---

```

@Bean
CommandLineRunner start(CompteRepository compteRepository, ClientRepository
    clientRepository, RepositoryRestConfiguration restConfiguration){
    return args -> {
        restConfiguration.exposeIdsFor(Compte.class);

        Client c1 = clientRepository.save(new Client(null, "Amal", null));
        Client c2 = clientRepository.save(new Client(null, "Ali", null));

        compteRepository.save(new Compte(null, Math.random() * 9000, new
            Date(), TypeCompte.EPARGNE, c1));
        compteRepository.save(new Compte(null, Math.random() * 9000, new
            Date(), TypeCompte.COURANT, c1));
        compteRepository.save(new Compte(null, Math.random() * 9000, new
            Date(), TypeCompte.EPARGNE, c2));

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

```

---



Cette méthode initialise deux clients (Amal et Ali) et leur associe plusieurs comptes avec des soldes aléatoires et des dates de création. L'utilisation de `CommandLineRunner` permet d'exécuter cette logique au démarrage de l'application.

#### Explications supplémentaires :

- `restConfiguration.exposeIdsFor(Compte.class)` : Permet d'exposer les IDs des entités `Compte` dans les réponses JSON, facilitant leur gestion côté client.
- Les comptes créés sont de types `EPARGNE` et `COURANT` et sont associés aux clients respectifs `c1` (Amal) et `c2` (Ali).
- L'initialisation est affichée dans la console pour vérification.

#### d. Utilisation des Liens REST pour Naviguer entre les Ressources

Avec Spring Data REST, les relations entre les entités sont automatiquement exposées sous forme de liens hypermédia dans les réponses JSON. Par exemple :

- `http://localhost:8082/api/clients/1/comptes` : Permet d'obtenir la liste des comptes associés au client ayant l'ID 1.
- `http://localhost:8082/api/comptes/1/client` : Permet d'obtenir les informations du client associé au compte ayant l'ID 1.



En utilisant les conventions de Spring Data REST, les relations entre les entités sont automatiquement exposées, facilitant ainsi la navigation entre les ressources sans nécessiter de code supplémentaire pour gérer les liens.

#### e. Personnalisation des Relations avec les Projections

Il est possible d'utiliser des projections pour personnaliser les informations affichées dans les relations. Par exemple, on peut afficher uniquement le nom et l'email d'un client lié à un compte en utilisant une projection.

```

@Projection(name = "clientDetails", types = Client.class)
public interface ClientProjection {

```

---

```
public String getNom();
public String getEmail();
}
```

---

Pour utiliser cette projection dans une requête, ajoutez le paramètre `projection=clientDetails` dans l'URL :

- `http://localhost:8082/api/comptes/1/client?projection=clientDetails` : Affiche uniquement le nom et l'email du client associé au compte.

**R** L'utilisation des projections permet de personnaliser les informations retournées dans les relations entre entités, offrant ainsi une plus grande flexibilité dans les réponses des APIs.

## 12. Requêtes de Recherche par Type de Compte

Spring Data REST permet d'ajouter des méthodes de recherche personnalisées dans les repositories. Dans cette section, nous allons créer une requête pour rechercher les comptes en fonction de leur type (EPARGNE ou COURANT).

### a. Définition de la Méthode de Recherche dans le CompteRepository

Pour ajouter une recherche par type, il suffit de définir une méthode dans l'interface `CompteRepository` en utilisant l'annotation `@RestResource`. Voici un exemple :

---

```
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;
import org.springframework.data.rest.core.annotation.RestResource;

@RepositoryRestResource
public interface CompteRepository extends JpaRepository<Compte, Long> {

    @RestResource(path = "/byType")
    public List<Compte> findByType(@Param("t") TypeCompte type);
}
```

---

**R** L'annotation `@RestResource` permet de personnaliser le chemin d'accès pour cette méthode de recherche. Dans cet exemple, `findByType` est accessible via l'URL `/api/comptes/search/byType`.

### b. Utilisation de la Requête de Recherche par Type dans l'URL

Une fois la méthode définie, elle peut être utilisée pour rechercher des comptes en spécifiant le type dans les paramètres de l'URL. Exemple :

- `http://localhost:8082/api/comptes/search/byType?t=EPARGNE` : Requête pour obtenir tous les comptes de type EPARGNE.
- `http://localhost:8082/api/comptes/search/byType?t=COURANT` : Requête pour obtenir tous les comptes de type COURANT.

**R** Cette méthode de recherche personnalisée permet aux utilisateurs de filtrer les comptes en fonction de leur type, ce qui améliore la flexibilité de l'API et facilite les recherches spécifiques dans les données.