

Architecture Distribuée et Middlewares

Mise en œuvre d'un micro-service

Compte rendu

Année Universitaire
2022/2023
GLSID2

Présenté par
AKASMIOU Ouassima

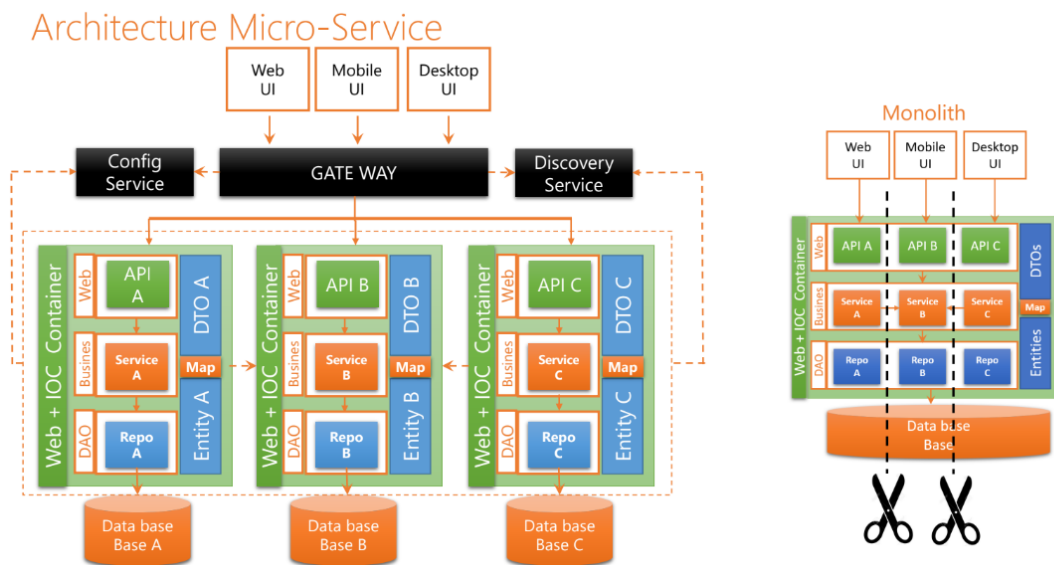
Encadré par
M. YOUSSEFI Mohammed



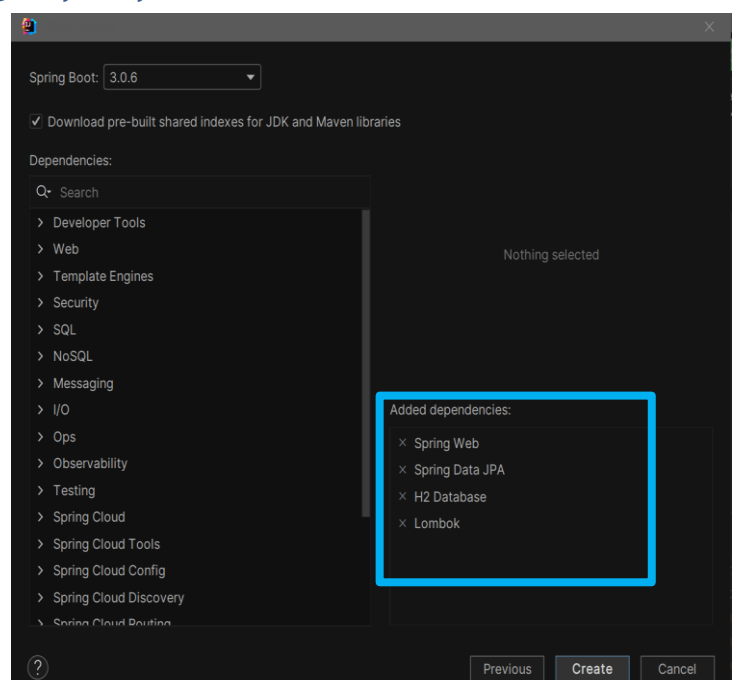
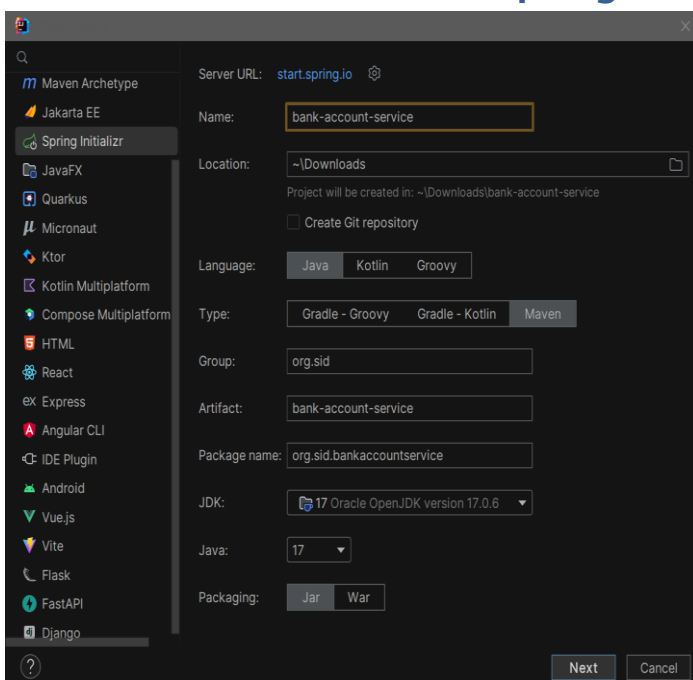
Ce projet consiste à développer un micro-service permettant de gérer efficacement des comptes bancaires.

• Architecture Micro-Service

L'architecture microservice est un style d'architecture logicielle qui consiste à diviser une application en un ensemble de services indépendants, déployables et évolutifs. Chaque service est développé, déployé et mis à l'échelle de manière autonome. Les microservices communiquent entre eux via des interfaces bien définies, souvent des API REST, et sont gérés par des outils d'orchestration tels que Kubernetes. Cette architecture permet une plus grande flexibilité, une évolutivité plus facile, une meilleure résilience et une maintenance plus aisée des applications.



• Création d'un projet Spring Boot avec les dépendances Web, Spring Data JPA, H2, Lombok



- Création de l'entité JPA BankAccount

```
20 usages
@Entity
@Data @NoArgsConstructor @AllArgsConstructor @Builder
public class BankAccount {
    no usages
    @Id
    private String id;
    no usages
    private Date createdAt;
    no usages
    private Double balance;
    no usages
    private String currency;
    no usages
    private AccountType type;
}
```

▼ entities

© BankAccount

Avec :

```
public enum AccountType {
    1 usage
    ⚡ CURRENT_ACCOUNT, SAVING_ACCOUNT
}
```

▼ enums

Ⓔ AccountType

- Création d'interface BankAccountRepository basée sur Spring Data

```
@Repository
public interface BankAccountRepository extends JpaRepository<BankAccount,String> {
}
```

▼ repositories

ⓘ BankAccountRepository

- Tester la couche DAO

BankAccountServiceApplication c'est une classe principale d'une application Spring Boot qui lance le micro-service BankAccountService. Il utilise l'annotation @SpringBootApplication pour configurer l'application. De plus, il implémente une méthode CommandLineRunner qui utilise la classe BankAccountRepository pour ajouter 10 comptes bancaires aléatoires dans la base de données au lancement de l'application.

```
@SpringBootApplication
public class BankAccountServiceApplication {
    no usages
    public static void main(String[] args) { SpringApplication.run(BankAccountServiceApplication.class, args); }
    no usages
    @Bean
    CommandLineRunner start(BankAccountRepository bankAccountRepository){
        return args -> {
            for (int i=0; i<10;i++) {
                BankAccount bankAccount = BankAccount.builder()
                    .id(UUID.randomUUID().toString())
                    .type(Math.random() > 0.5 ? AccountType.CURRENT_ACCOUNT : AccountType.SAVING_ACCOUNT)
                    .balance(10000+Math.random()*90000)
                    .createdAt(new Date())
                    .currency("MAD")
                    .build();
                bankAccountRepository.save(bankAccount);
            }
        };
    }
}
```

- Création de Web service Restfull qui permet de gérer des comptes

Alors on allons créer un contrôleur REST qui va gérer les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer) sur une entité BankAccount, stockée dans une base de données via un objet BankAccountRepository. Les méthodes GET, POST, PUT et DELETE sont utilisées pour récupérer, enregistrer, mettre à jour et supprimer des enregistrements BankAccount dans la base de données.

```
@RestController
public class AccountRestController {

    7 usages
    private BankAccountRepository bankAccountRepository;

    no usages
    public AccountRestController(BankAccountRepository bankAccountRepository) {
        this.bankAccountRepository = bankAccountRepository;
    }

    no usages
    @GetMapping("Ⓢ"/bankAccounts")
    public List<BankAccount> bankAccounts() { return bankAccountRepository.findAll(); }

    no usages
    @GetMapping("Ⓢ"/bankAccounts/{id}")
    public BankAccount bankAccount(@PathVariable String id) {
        return bankAccountRepository.findById(id).orElseThrow(() -> new RuntimeException(String.format("Account %s not found", id)));
    }

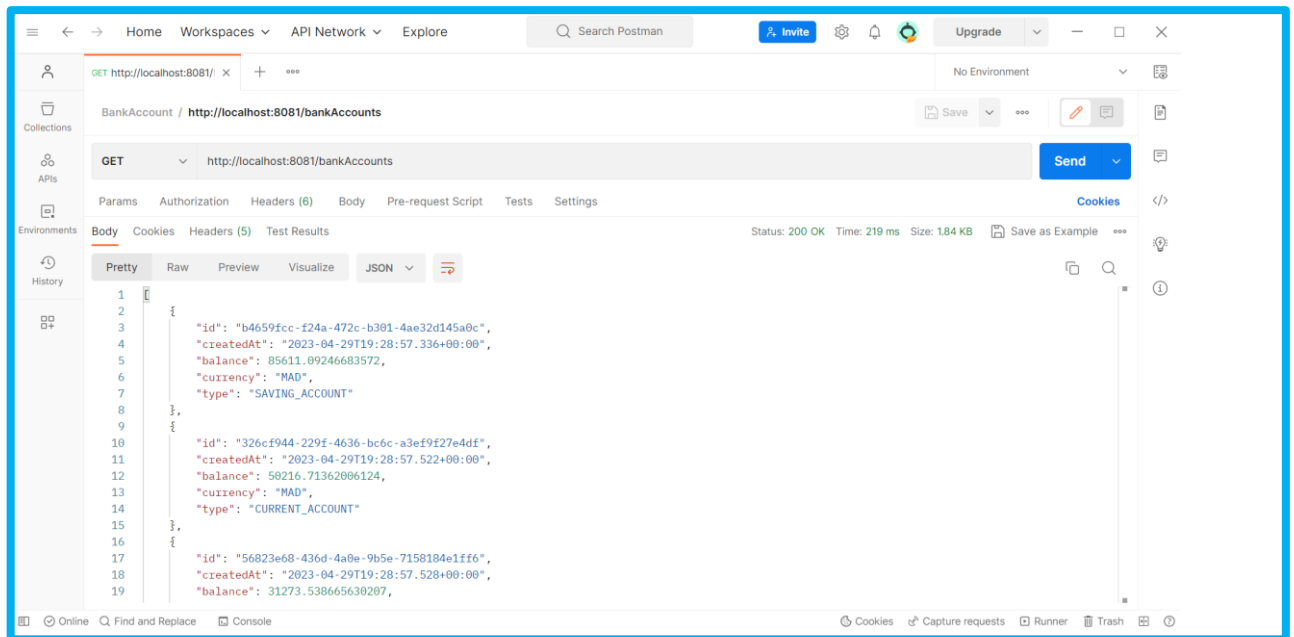
    no usages
    @PostMapping("Ⓢ"/bankAccounts")
    public BankAccount save(@RequestBody BankAccount bankAccount) {
        if(bankAccount.getId() == null) bankAccount.setId(UUID.randomUUID().toString());
        return bankAccountRepository.save(bankAccount);
    }

    no usages
    @PutMapping("Ⓢ"/bankAccounts/{id}")
    public BankAccount update(@PathVariable String id, @RequestBody BankAccount bankAccount) {
        BankAccount account=bankAccountRepository.findById(id).orElseThrow();
        if (bankAccount.getBalance() != null) account.setBalance(bankAccount.getBalance());
        if (bankAccount.getCreatedAt() != null) account.setCreatedAt(new Date());
        if (bankAccount.getType() != null) account.setType(bankAccount.getType());
        if (bankAccount.getCurrency() != null) account.setCurrency(bankAccount.getCurrency());
        return bankAccountRepository.save(account);
    }

    no usages
    @DeleteMapping("Ⓢ"/bankAccounts/{id}")
    public void deleteAccount(@PathVariable String id){
        bankAccountRepository.deleteById(id);
    }
}
```

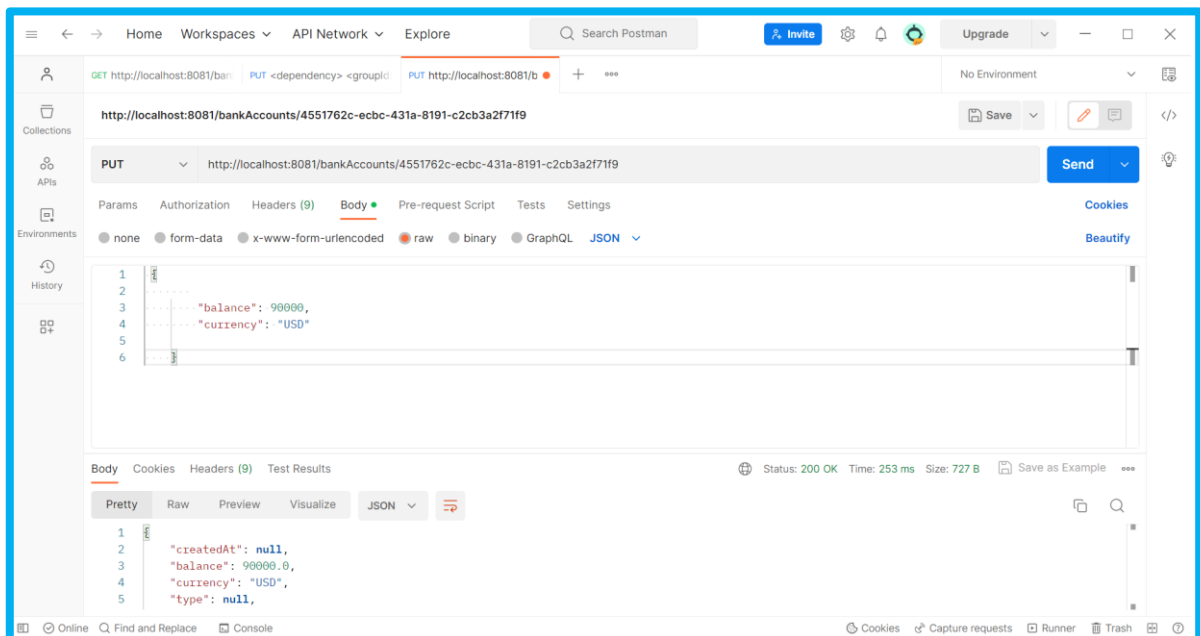
- **Test de web micro-service en utilisant un client REST comme Postman**

Ici nous allons Tester l'API de l'endpoint `"/bankAccounts"` en utilisant une requête GET avec un client REST tel que Postman, puis on va vérifier si la liste des comptes bancaires est retournée avec succès.



Maintenant, nous allons tester la méthode "update" d'un compte ayant l'identifiant "4551762c-ecbc-431a-8191-c2cb3a2f71f9".

Nous utiliserons Postman, un client REST, pour envoyer une requête "PUT" à l'URL "`http://localhost:8080/bankAccounts/4551762c-ecbc-431a-8191-c2cb3a2f71f9`", avec un corps de requête contenant les nouvelles informations du compte à mettre à jour.



- **Génération et test de la documentation Swagger pour les API REST du service web.**

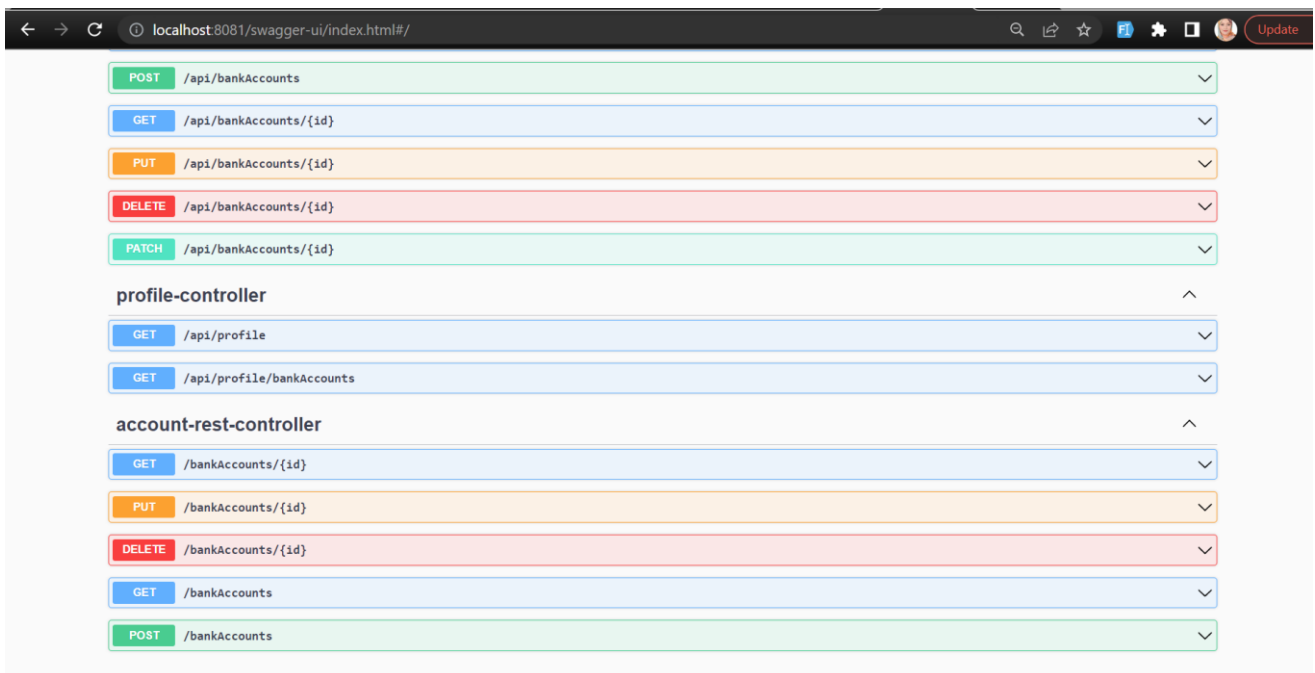
La documentation **Swagger** est un outil utile pour les développeurs et les consommateurs de services web, car elle fournit une documentation claire et précise sur les différentes API REST disponibles, leurs entrées et sorties attendues, ainsi que les erreurs possibles qui peuvent être rencontrées.

Pour générer et tester la documentation Swagger pour les API REST du service web, vous pouvez suivre les étapes suivantes :

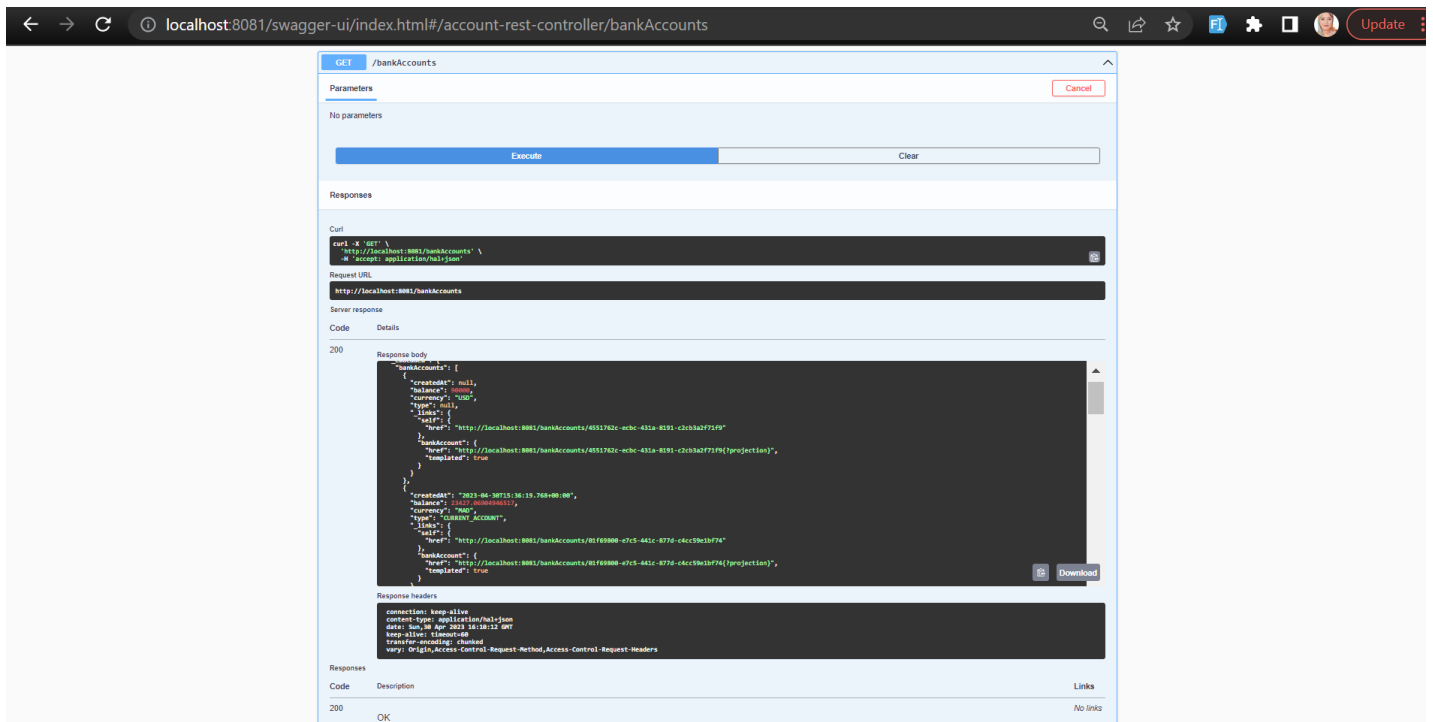
- Ajouter la dépendance Swagger à votre projet (pom.xml).

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.0.4</version>
</dependency>
```

- Lancer votre service web et accéder à la documentation Swagger générée en utilisant l'URL appropriée. Comme dans notre projet nous allons utiliser l'URL : <http://localhost:8081/swagger-ui.html>



Nous allons utiliser Swagger pour tester la méthode GET de notre API REST. Nous allons envoyer une requête à l'URL correspondante et vérifier que les données retournées sont bien celles attendues. En utilisant Swagger, nous pourrions également visualiser la documentation de notre API et découvrir les différentes méthodes disponibles.

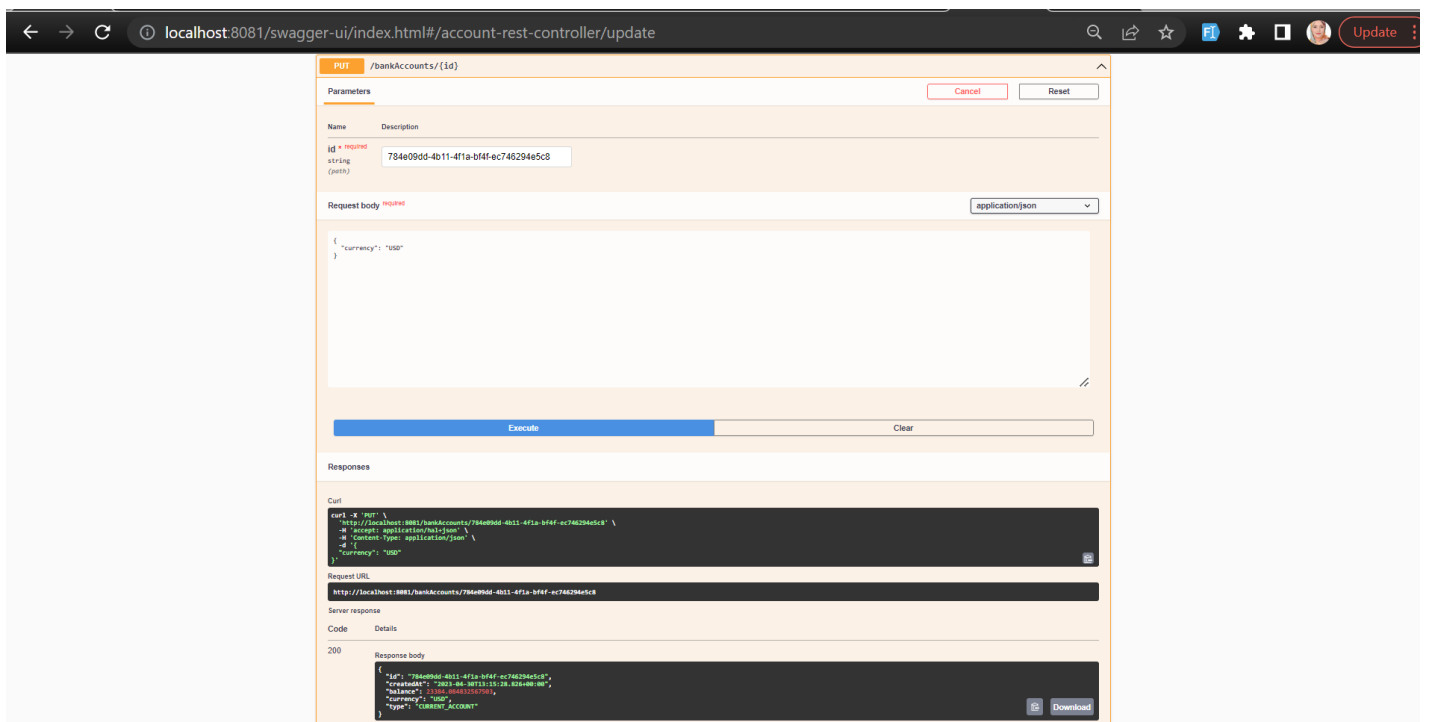


The screenshot shows the Swagger UI for the endpoint `GET /bankAccounts`. The interface includes a "Parameters" section with "No parameters", an "Execute" button, and a "Responses" section. The response body is displayed as a JSON array of three bank account objects. The first object has a balance of 1000, the second has a balance of 2000, and the third has a balance of 3000. The response headers show a 200 status code and a "Content-Type" of "application/json".

```
curl -X GET \
  "http://localhost:8081/swagger-ui/index.html#/account-rest-controller/bankAccounts" \
  -H "accept: application/json"
```

```
{
  "accounts": [
    {
      "id": "784e09dd-4b11-4f1a-bf4f-ec746294e5c8",
      "created_at": "2023-04-20T15:16:19.760+00:00",
      "balance": 1000,
      "currency": "USD",
      "type": "CURRENT_ACCOUNT",
      "link": "http://localhost:8081/swagger-ui/index.html#/account-rest-controller/bankAccounts/{id}"
    },
    {
      "id": "784e09dd-4b11-4f1a-bf4f-ec746294e5c9",
      "created_at": "2023-04-20T15:16:19.760+00:00",
      "balance": 2000,
      "currency": "USD",
      "type": "CURRENT_ACCOUNT",
      "link": "http://localhost:8081/swagger-ui/index.html#/account-rest-controller/bankAccounts/{id}"
    },
    {
      "id": "784e09dd-4b11-4f1a-bf4f-ec746294e5ca",
      "created_at": "2023-04-20T15:16:19.760+00:00",
      "balance": 3000,
      "currency": "USD",
      "type": "CURRENT_ACCOUNT",
      "link": "http://localhost:8081/swagger-ui/index.html#/account-rest-controller/bankAccounts/{id}"
    }
  ]
}
```

Maintenant, nous allons tester la méthode PUT du service web en utilisant Swagger.



The screenshot shows the Swagger UI for the endpoint `PUT /bankAccounts/{id}`. The interface includes a "Parameters" section with a required string parameter `id` with the value `784e09dd-4b11-4f1a-bf4f-ec746294e5c8`. The "Request body" section shows a JSON object with a `currency` field set to `USD`. The "Execute" button is visible. The response body is displayed as a JSON object representing the updated bank account.

```
curl -X PUT \
  "http://localhost:8081/swagger-ui/index.html#/account-rest-controller/update/{id}" \
  -H "accept: application/json" \
  -H "content-type: application/json" \
  -d '{
    "currency": "USD"
  }'
```

```
{
  "id": "784e09dd-4b11-4f1a-bf4f-ec746294e5c8",
  "created_at": "2023-04-20T15:16:19.760+00:00",
  "balance": 1000,
  "currency": "USD",
  "type": "CURRENT_ACCOUNT",
  "link": "http://localhost:8081/swagger-ui/index.html#/account-rest-controller/bankAccounts/{id}"
}
```


- Exposition d'une API RESTful à l'aide de Spring Data Rest en utilisant des projections pour sélectionner les données à inclure dans la réponse.

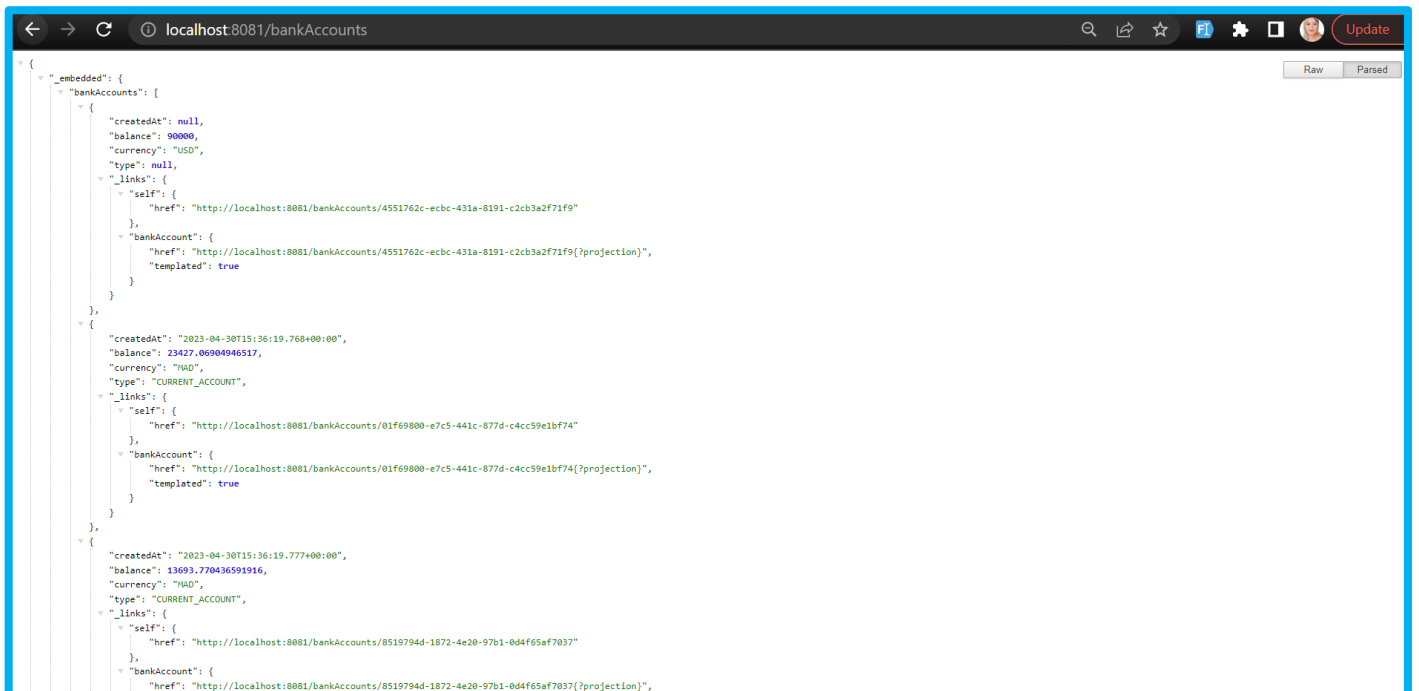
Premièrement nous allons ajouter cette dépendance de spring data rest dans le fichier pom.xml de notre projet.

```
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

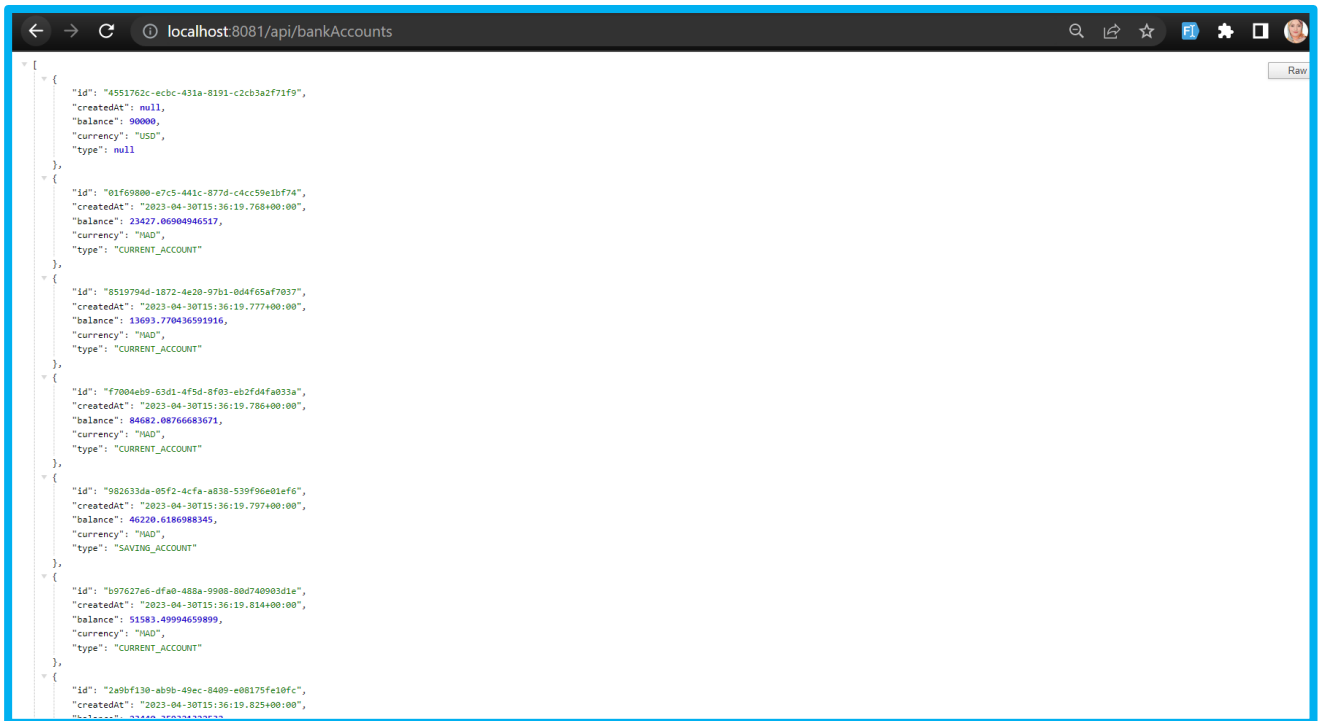
On utilise l'annotation `@RequestMapping("/api")` pour différencier l'URL de l'API RESTful standard de celle utilisant Spring Data Rest.

```
no usages
@RestController
@RequestMapping("/api")
public class AccountRestController {
```

L'URL de l'API RESTful en utilisant Spring Data Rest :



L'URL de l'API RESTful standard :



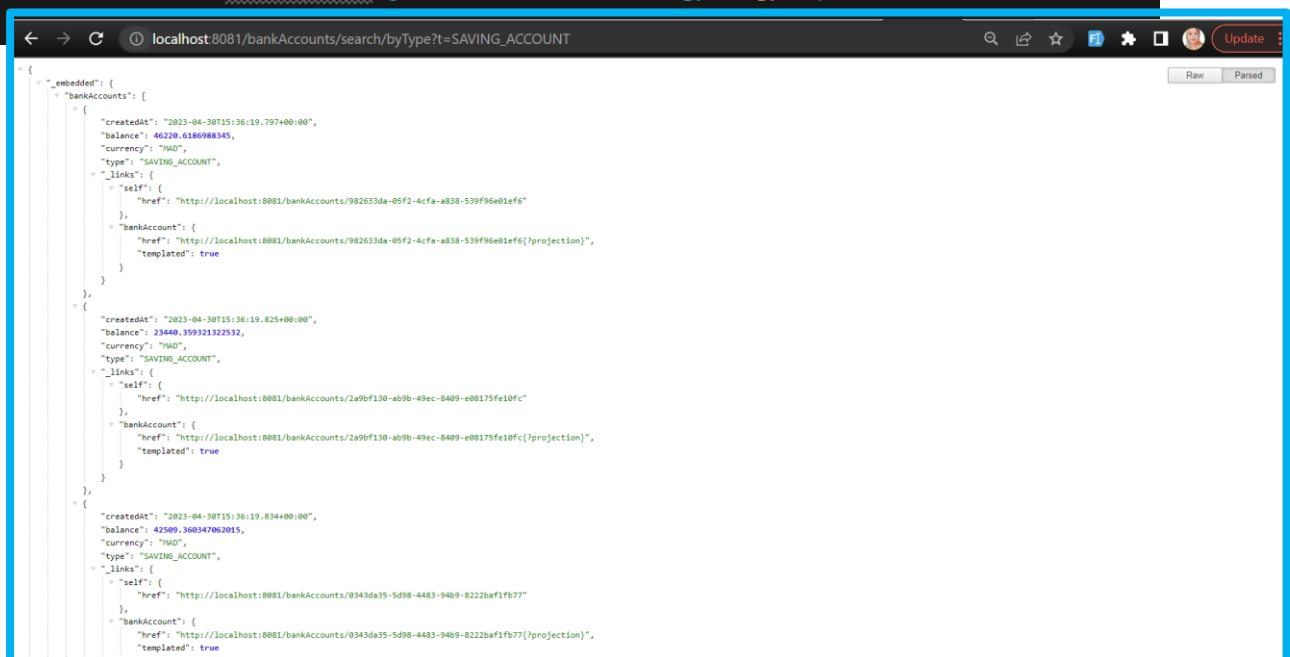
La méthode `findByType()` est une méthode personnalisée définie dans cette interface, qui utilise l'annotation `@RestResource` pour personnaliser le chemin d'accès de l'API ("/byType") et rendre cette méthode accessible via HTTP GET. Elle prend un paramètre de type `AccountType` et retourne une liste de `BankAccount` correspondant à ce type.

```

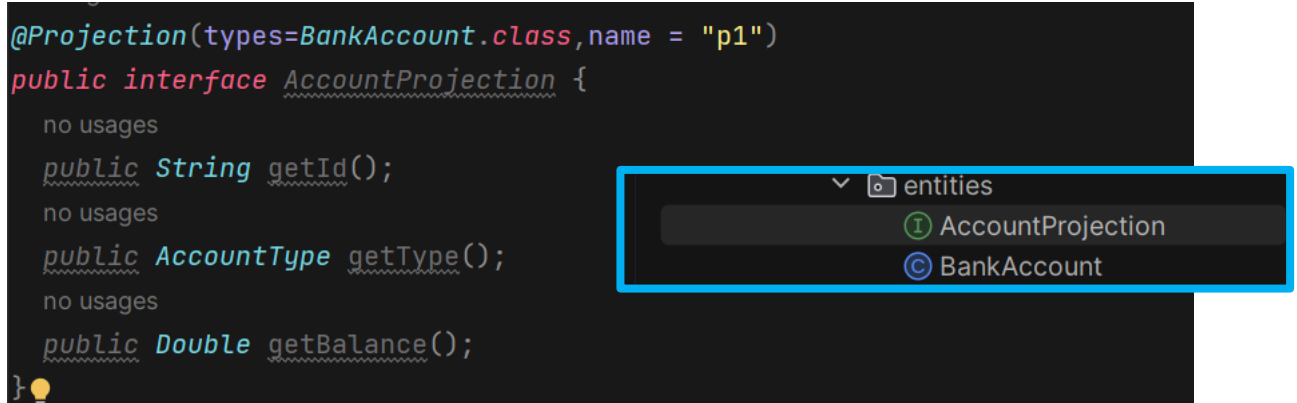
@RepositoryRestResource
public interface BankAccountRepository extends JpaRepository<BankAccount,String> {
    no usages

    @RestResource(path = "/byType")
    List<BankAccount> findByType(@Param("t") AccountType type);
}

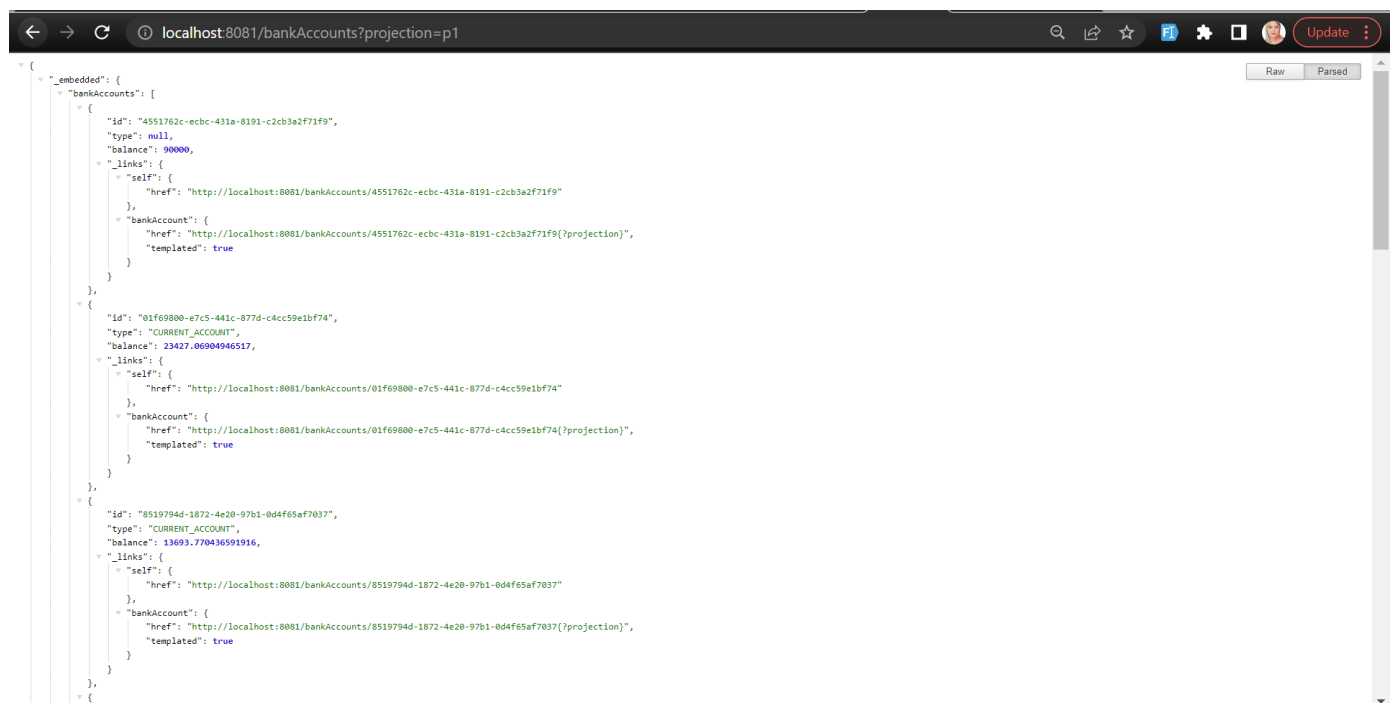
```



On définit une projection pour la classe BankAccount, avec le nom "p1". La projection inclut uniquement les méthodes d'accès getId(), getType() et getBalance().



Cette URL <http://localhost:8081/bankAccounts?projection=p1> permet de récupérer tous les comptes bancaires avec les informations spécifiques de la projection "p1"



• Création des DTOs et Mappers

Les DTO (Data Transfer Object) et les mappers sont utilisés pour séparer la logique métier de la couche de présentation dans une application. Les DTO permettent de transférer uniquement les données nécessaires entre les différentes couches de l'application, tandis que les mappers sont utilisés pour convertir les objets entre différents formats (par exemple, entre les objets de la couche métier et les DTO de la couche de présentation). Cela permet de réduire la complexité et de faciliter la maintenance de l'application, en réduisant les dépendances entre les différentes couches et en permettant une plus grande flexibilité dans l'évolution de l'application.

▼ dto

- ⦿ BankAccountRequestDTO
- ⦿ BankAccountResponseDTO

6 usages

```
@Data @AllArgsConstructor @NoArgsConstructor @Builder
public class BankAccountRequestDTO {
    no usages
    private Double balance;
    no usages
    private String currency;
    no usages
    private AccountType type;
}
```

11 usages

```
@Data @AllArgsConstructor @NoArgsConstructor @Builder
public class BankAccountResponseDTO {
    no usages
    private String id;
    no usages
    private Date createdAt;
    no usages
    private Double balance;
    no usages
    private String currency;
    no usages
    private AccountType type;
}
```

▼ mappers

- ⦿ AccountMapper

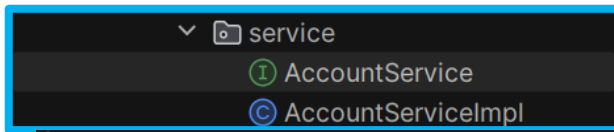
@Component

```
public class AccountMapper {
```

1 usage

```
public BankAccountResponseDTO fromBankAccount(BankAccount bankAccount){
    BankAccountResponseDTO bankAccountResponseDTO=new BankAccountResponseDTO();
    BeanUtils.copyProperties(bankAccount,bankAccountResponseDTO);
    return bankAccountResponseDTO;
}
```

- Création de la couche Service (métier) et du micro service



4 suggestions 1 implementation

```
public interface AccountService {
```

1 usage 1 implementation

```
    public BankAccountResponseDTO addAccount(BankAccountRequestDTO bankAccountDTO);
}
```

```
public class AccountServiceImpl implements AccountService {
```

1 usage

@Autowired

```
private BankAccountRepository bankAccountRepository;
```

1 usage

@Autowired

```
private AccountMapper accountMapper;
```

1 usage

@Override

```
public BankAccountResponseDTO addAccount(BankAccountRequestDTO bankAccountDTO) {
```

```
    BankAccount bankAccount=BankAccount.builder()
```

```
        .id(UUID.randomUUID().toString())
```

```
        .createdAt(new Date())
```

```
        .balance(bankAccountDTO.getBalance())
```

```
        .type(bankAccountDTO.getType())
```

```
        .currency(bankAccountDTO.getCurrency())
```

```
        .build();
```

```
    BankAccount saveBankAccount=bankAccountRepository.save(bankAccount);
```

```
    BankAccountResponseDTO bankAccountResponseDTO=accountMapper.fromBankAccount(saveBankAccount);
```

```
    return bankAccountResponseDTO;
```

```
}
```

On modifier la méthode save

```
@PostMapping("/bankAccounts")
```

```
public BankAccountResponseDTO save(@RequestBody BankAccountRequestDTO requestDTO) {
```

```
    return accountService.addAccount(requestDTO);
```

```
}
```

FIN