

Rapport Revolut Miage

Clément Mercier

Année 2021-2022

**Projet réalisé dans le cadre du module
« Passage à l'échelle »**

Encadrant universitaire : Olivier Perrin

1) Introduction

Dans le cadre du projet pour le module passage à l'échelle, il nous a été demandé de réaliser une API pour la gestion de compte bancaire permettant à des utilisateurs inscrits de gérer un compte bancaire, à la façon de Revolut.

2) Travail réalisé

a) Technologies utilisées

i. Java Spring

C'est un Framework open source pour construire et définir l'infrastructure d'une application Java avec de nombreuses fonctionnalités permettant de faciliter le développement et les tests. L'injection de dépendances largement utilisé pour la réalisation de se projet fait la force de Spring. La facilité pour créer des tests est également appréciable quand on souhaite avoir un code solide.

ii. Spring Boot

Spring Boot est également un Framework qui s'appuie sur Spring en y rajoutant des fonctions orientées développement web comme les standards REST avec toute la configuration déjà faite ou encore le serveur permettant en un clic d'avoir une application qui tourne sans difficulté (c.f M Blanchard pour plus de complexité avec JEE). Etant donné que le projet porte sur une API, Spring Boot répond parfaitement à cette problématique.

iii. Postman

C'est une application qui permet de faire des requêtes HTTP avec n'importe quelle méthode en y incluant des headers et un body. Etant donné qu'il n'y a pas de frontend dans le projet, Postman reste la solution la plus intéressante pour tester son API. Je m'en suis servi pour stocker tous mes « end points » afin de les tester avec une variable global contenant un token automatiquement ajouté à l'authentification d'un utilisateur

iv. Trello

C'est une application web permettant de gérer son développement en y ajoutant des cartes représentant les fonctionnalités à faire dans l'application avec tout un cycle de vie (SCRUM). Le tableau est accessible ici : <https://trello.com/b/HGPKFkNT/revolutmiage>

v. Github

Voici le lien du repository git de mon projet :
<https://github.com/Clems54/revolutmiage>

b) Dépendances du projet

Pour la création du projet et l'ajout de toutes les dépendances nécessaires à la réalisation de celui-ci, j'ai utilisé le site « Spring Initializr » pour générer un projet initial avec un pom.xml fonctionnel. On y retrouve « Lombok » qui est une librairie permettant de réduire le code des classes en y ajoutant des annotations qui généreront des getters/setters ou encore des constructeurs à la compilation. Spring Boot a également été ajouté pour inclure toutes les fonctionnalités nécessaires à la création de l'API. La dépendance JPA a également été ajoutée pour faciliter la gestion de la persistance des entités au sein de l'application. C'est un ORM. D'autres librairies ont également été ajoutées pour faciliter l'écriture des tests, la validation des éléments fournis par l'API ou encore la structuration des requêtes renvoyées par notre application (Hateoas). Le projet tourne avec une base de données H2 pour gagner du temps. À terme il sera forcément nécessaire de passer par une BDD persistante comme PostgreSQL.

c) Architecture du projet

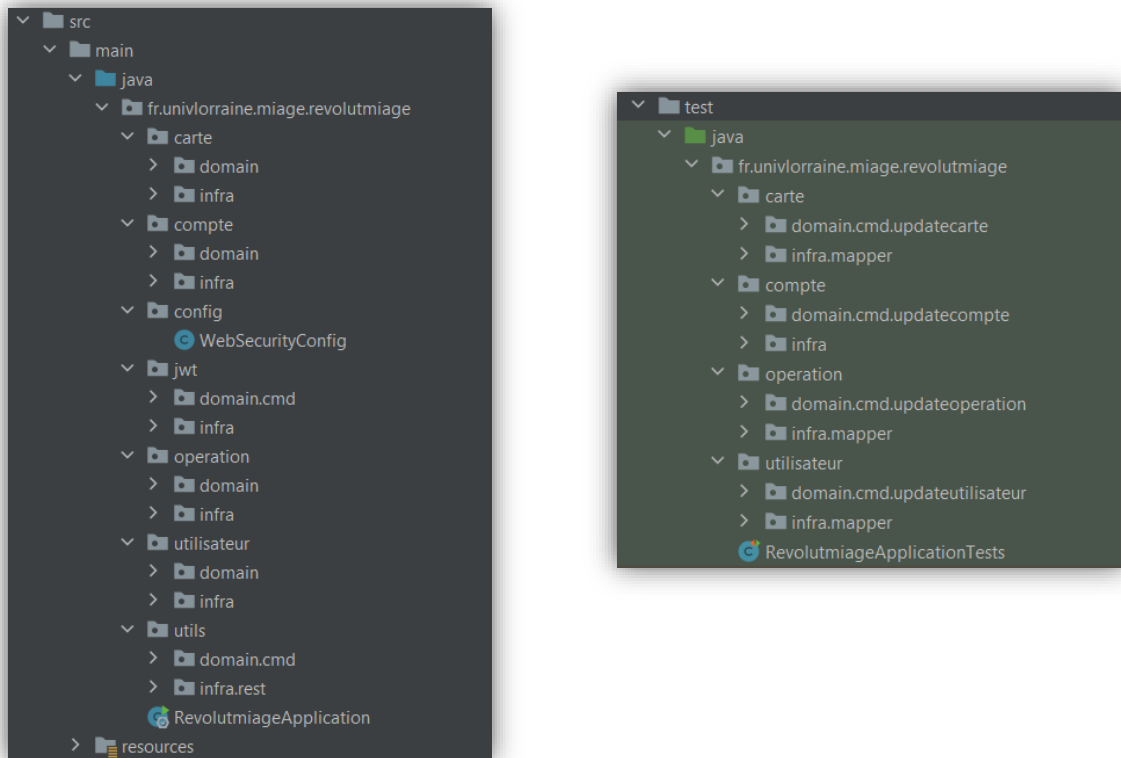


Figure 1 : Architecture du projet

L'outil de Spring pour générer un projet a déjà fixé l'architecture de base du projet avec un « src » contenant tout le code du projet. Sous ce répertoire on a « main » qui concerne le code de l'application et « test » qui, comme son nom l'indique contient les tests. Dans le code de l'application, on retrouve une partie « java » avec les classes Java et « resources » contenant les ressources de l'application comme les propriétés/configuration ou encore la gestion de la base de données.

Dans le but de faire une application des plus solides possible, j'ai choisi de mettre en place une architecture hexagonale. J'espère en respecter tous les codes parce qu'on n'en a pas encore trop fait mais je trouve tout de même puissant de pouvoir séparer le code métier de l'architecture pour que l'application soit modifiable, en termes de technologies utilisées sans avoir un impact significatif sur le temps à y passer.

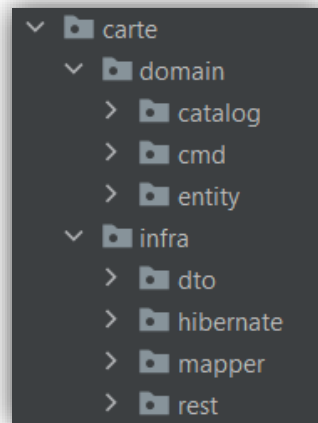


Figure 2 : Structure de l'Architecture hexagonale de la carte

Le plus difficile aura été de définir ce qui fait partie du domaine et ce qui fait partie de l'infrastructure. De ce que j'ai compris, le domaine contient les éléments qui sont indépendants de toutes technologies. Comme vous pouvez le voir dans la Figure 2 il y a l'entité carte défini dans le domaine car peu importe la technologie utilisée, l'entité ne va jamais être modifiée. Ces attributs ne bougeront pas. Sauf si bien entendu, une nouvelle fonctionnalité amène à l'ajout ou la suppression d'un champ.

Dans l'architecture hexagonale, il y a également une autre règle importante : L'infrastructure peut dépendre du domaine mais jamais l'inverse. Ce qui signifie qu'une classe du domaine peut être appelé dans l'infrastructure mais une classe de l'infrastructure ne peut jamais être appelé dans le domaine. Pour que cela soit possible, j'ai utilisé ce que nous offre de mieux Spring : « L'injection de dépendances » !

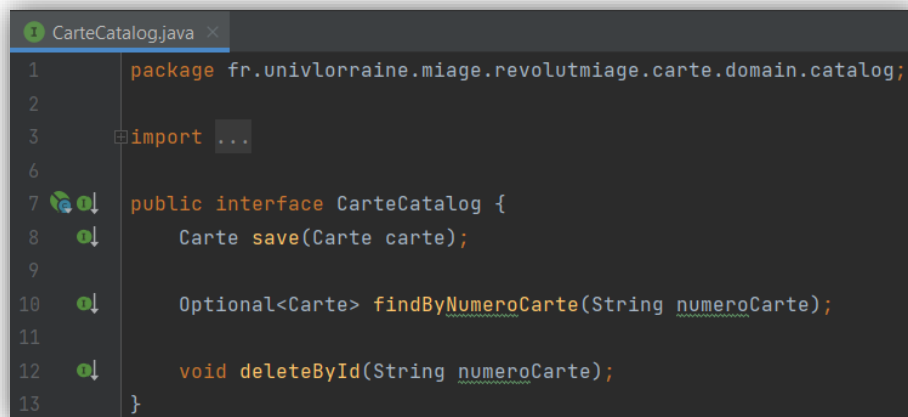
Dans l'infrastructure, j'ai ajouté l'interface de JPA sous le package Hibernate.

```
1 package fr.univlorraine.miage.revolutmiage.carte.infra.hibernate;
2
3 import ...
4
5
6
7
8 public interface CarteRepository extends CrudRepository<Carte, String> {
9     Optional<Carte> findByNumeroCarte(String numeroCarte);
10
11     void deleteByNumeroCarte(String numeroCarte);
12 }
```

Figure 3 : Interface carte de JPA

Dans le package « cmd » du domaine, il y a les classes qui permettent la validation des différents appels à l'API. Mais certaines validations passent par un appel en base pour vérifier l'existence d'une entité. Par exemple, lors de la création d'une carte, il faut vérifier si le compte assigné à celle-ci existe. Malheureusement, pour se faire, il faudrait faire appel au JPA présent dans l'infrastructure mais comme dit précédemment, on ne peut pas appeler l'infrastructure depuis le domaine. C'est là que l'injection de dépendance opère. Dans le

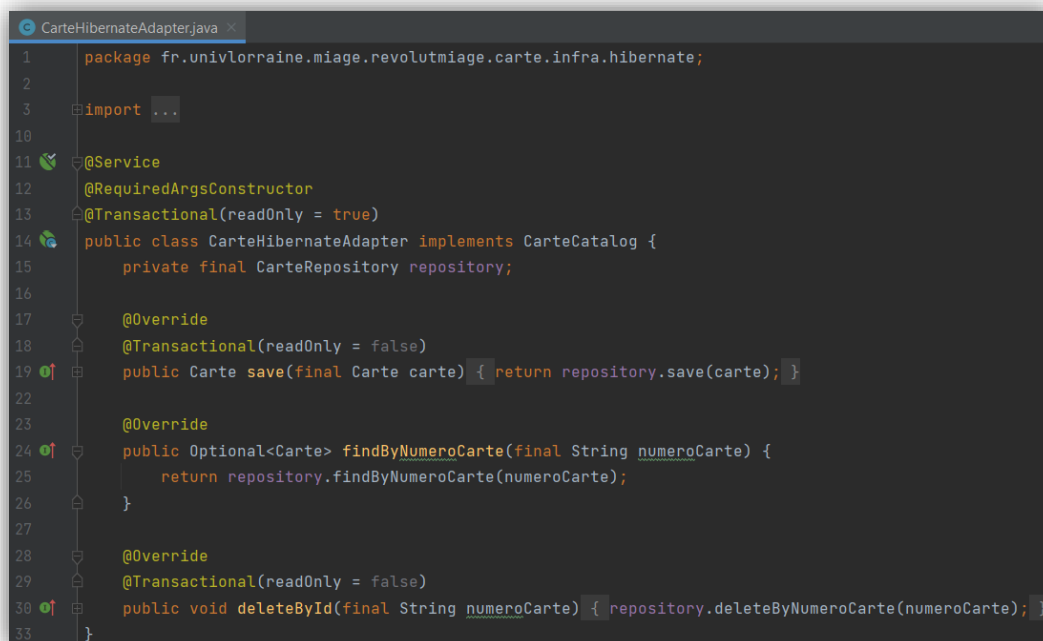
domaine, j'ai créé un package « catalog » contenant les interfaces des opérations pouvant être effectuées sur les entités de l'application.

A screenshot of an IDE showing the code for CarteCatalog.java. The code defines a package, imports, and a public interface CarteCatalog with three methods: save, findByNumeroCarte, and deleteById.

```
1 package fr.univlorraine.miage.revolutmiage.carte.domain.catalog;
2
3 import ...
4
5
6
7 public interface CarteCatalog {
8     Carte save(Carte carte);
9
10    Optional<Carte> findByNumeroCarte(String numeroCarte);
11
12    void deleteById(String numeroCarte);
13 }
```

Figure 4 : Catalogue de l'entité carte

Ces interfaces sont implémentées par des classes adapteurs pour JPA en tant que service. En appelant le catalogue, il est donc possible d'accéder aux méthodes exclusives à JPA.

A screenshot of an IDE showing the code for CarteHibernateAdapter.java. The code defines a package, imports, and a public class CarteHibernateAdapter that implements the CarteCatalog interface. It uses JPA annotations like @Service, @RequiredArgsConstructor, @Transactional, and @Override.

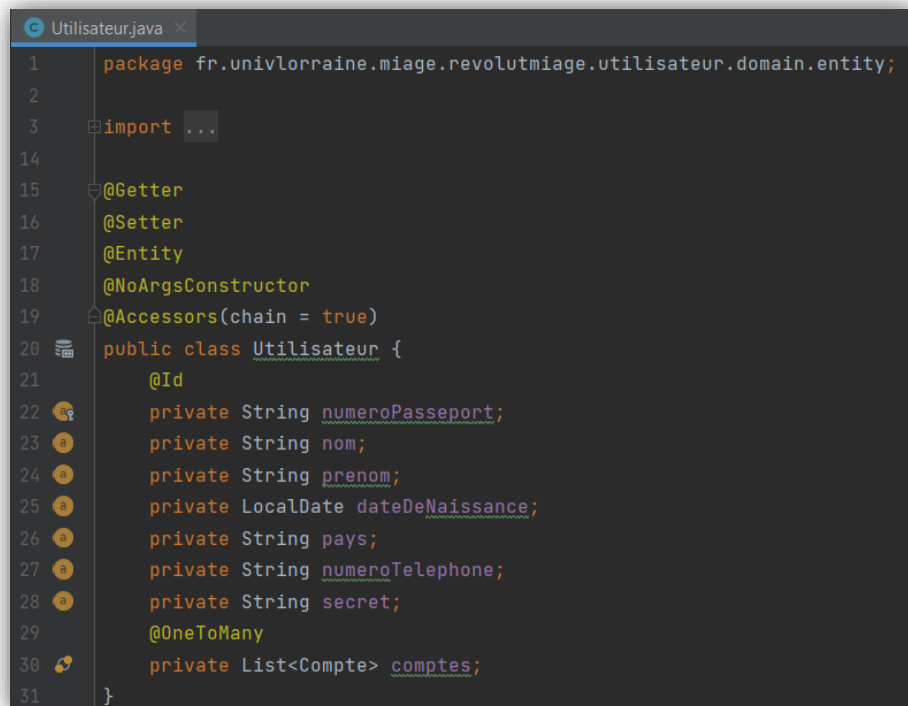
```
1 package fr.univlorraine.miage.revolutmiage.carte.infra.hibernate;
2
3 import ...
4
5
6
7
8
9
10
11 @Service
12 @RequiredArgsConstructor
13 @Transactional(readOnly = true)
14 public class CarteHibernateAdapter implements CarteCatalog {
15     private final CarteRepository repository;
16
17     @Override
18     @Transactional(readOnly = false)
19     public Carte save(final Carte carte) { return repository.save(carte); }
20
21
22
23     @Override
24     public Optional<Carte> findByNumeroCarte(final String numeroCarte) {
25         return repository.findByNumeroCarte(numeroCarte);
26     }
27
28     @Override
29     @Transactional(readOnly = false)
30     public void deleteById(final String numeroCarte) { repository.deleteByNumeroCarte(numeroCarte); }
31
32
33 }
```

Figure 5 : Adapteur entre JPA et le catalogue carte

d) Le format des réponses

Dans un premier temps pour les réponses des requêtes à l'API, je renvoie toujours un DTO pour être sûr que les entités soient toujours bien sérialisées. Ce qui veut dire que pour chaque entité, un DTO est créé.

Pour exemple, l'entité Utilisateur contient un attribut « dateDeNaissance » avec un type « LocalDate » et on ne sait pas trop comment celui-ci va être sérialisé dans la réponse donc l'utilisation d'un DTO peut être intéressant.



```
1 package fr.univlorraine.miage.revolutmiage.utilisateur.domain.entity;
2
3 import ...
4
14
15 @Getter
16 @Setter
17 @Entity
18 @NoArgsConstructor
19 @Accessors(chain = true)
20 public class Utilisateur {
21     @Id
22     private String numeroPasseport;
23     private String nom;
24     private String prenom;
25     private LocalDate dateDeNaissance;
26     private String pays;
27     private String numeroTelephone;
28     private String secret;
29     @OneToMany
30     private List<Compte> comptes;
31 }
```

Figure 6 : Entité Utilisateur

Pour passer d'une entité à son DTO il est nécessaire d'utiliser un « mappeur ». Celui-ci reprend tous les attributs de l'entité pour les convertir dans des types simples. Comme on peut le voir dans la Figure 6, il y a un attribut contenant la liste des comptes et cela peut devenir vite chronophage de parcourir les attributs de chaque compte pour les mapper en DTO. Si une modification doit être apportée à la classe compte, il serait nécessaire de la répercuter partout. C'est pour cela qu'il est possible de définir un DTO comme type « simple » pour cette liste, ce qui donnerait une liste de DTO de compte à la sortie du mappeur. Le mappeur peut être rébarbatif à mettre en place pour toutes les entités car c'est toujours la même opération à faire : récupérer tous les attributs et les convertir s'il le faut pour créer le DTO. Mais grâce à MapStruct, il n'est pas forcément nécessaire de le faire, il suffit de créer une interface en y ajoutant les bonnes annotations pour qu'à la compilation du projet, Spring génère l'implémentation de cette interface pour nous éviter de le faire nous-même.

```

1 package fr.univlorraine.miage.revolutmiage.Utilisateur.infra.mapper;
2
3 import ...
4
5
6
7
8
9
10
11 @Mapper(uses = {CompteMapper.class}, componentModel = "spring", imports = LocalDate.class)
12 public interface UtilisateurMapper {
13
14     @Mapping(target = "dateDeNaissance", expression = "java(entity.getDateDeNaissance().toString())")
15     UtilisateurDTO toDto(Utilisateur entity);
16 }

```

Figure 7 : Interface Mapper de l'utilisateur

Comme vous pouvez le voir dans la *Figure 7*, il est possible de fixer les attentes concernant la conversion d'un type complexe dans une annotation directement. Il suffit ensuite de simplement packager le projet pour que l'implémentation soit créée. Il suffira par la suite d'appeler cette interface dans le code et l'injection de dépendances fera le reste en allant directement chercher l'implémentation de votre mappeur dans le projet. Moins il y a de code à écrire et moins il y a de bug.

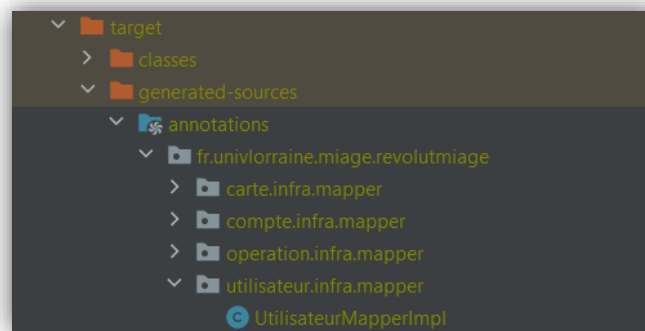


Figure 8 : Emplacement du mappeur Utilisateur généré


```

1 package fr.univlorraine.miage.revolutmiage.utilisateur.infra.mapper;
2
3 import ...
4
5 @Generated(
6     value = "org.mapstruct.ap.MappingProcessor",
7     date = "2021-12-30T11:46:36+0100",
8     comments = "version: 1.4.2.Final, compiler: javac, environment: Java 17.0.1 (Oracle Corporation)"
9 )
10 @Component
11 public class UtilisateurMapperImpl implements UtilisateurMapper {
12
13     @Autowired
14     private CompteMapper compteMapper;
15
16     @Override
17     public UtilisateurDTO toDto(Utilisateur entity) {
18         if ( entity == null ) {
19             return null;
20         }
21
22         UtilisateurDTO utilisateurDTO = new UtilisateurDTO();
23
24         utilisateurDTO.setNom( entity.getNom() );
25         utilisateurDTO.setPrenom( entity.getPrenom() );
26         utilisateurDTO.setPays( entity.getPays() );
27         utilisateurDTO.setNumeroPasseport( entity.getNumeroPasseport() );
28         utilisateurDTO.setNumeroTelephone( entity.getNumeroTelephone() );
29         utilisateurDTO.setComptes( compteMapper.toDtos( entity.getComptes() ) );
30
31         utilisateurDTO.setDateDeNaissance( entity.getDateDeNaissance().toString() );
32
33         return utilisateurDTO;
34     }
35 }

```

Figure 9 : Mappeur Utilisateur généré

Il est également possible de tester ces mappeurs pour être certains qu'il n'y a pas un souci sur la génération de l'implémentation. Il suffit d'appeler l'interface du mappeur à tester et de vérifier si les valeurs du DTO sont celles attendues.

```

@Test
@DisplayName("Devrait retourner l'obj en DTO")
void toDto() {
    // GIVEN
    final Utilisateur obj = new Utilisateur()
        .setNom(NOM)
        .setPrenom(PRENOM)
        .setPays(PAYS)
        .setNumeroPasseport(NUMERO_PASSEPORT)
        .setDateDeNaissance(DATE_DE_NAISSANCE)
        .setNumeroTelephone(NUMERO_TELEPHONE)
        .setSecret(SECRET)
        .setComptes(COMPTES);

    // WHEN
    final UtilisateurDTO actual = subject.toDto(obj);

    // THEN
    Assertions.assertEquals(NOM, actual.getNom());
    Assertions.assertEquals(PRENOM, actual.getPrenom());
    Assertions.assertEquals(PAYS, actual.getPays());
    Assertions.assertEquals(NUMERO_PASSEPORT, actual.getNumeroPasseport());
    Assertions.assertEquals(DATE_DE_NAISSANCE, LocalDate.parse(actual.getDateDeNaissance()));
    Assertions.assertEquals(NUMERO_TELEPHONE, actual.getNumeroTelephone());
    Assertions.assertEquals(COMPTES.size(), actual.getComptes().size());
}

```

Figure 10 : Test du mappage d'une entité Utilisateur en DTO

e) La validation

Dans une API, le plus important est de vérifier toutes les données qui sont envoyés par un utilisateur, le but étant de ne faire confiance à personne et de faire valider la moindre saisie pour ne pas avoir de soucis. Il a donc été nécessaire que je mette en place un moyen de valider les saisies de façon simple.

Pour la création d'un utilisateur, je récupère le body du POST sous forme d'un Input. J'envoie ensuite cet Input dans une classe qui va gérer la validation.

```
@PostMapping
public ResponseEntity<> creerUtilisateur(@RequestBody final UpdateUtilisateurInput input) {
    updateUtilisateur.accept(input.setCreation(true));
    return ResponseEntity.created(
        linkTo(Utilisateur.class).slash(input.getNumeroPasseport()).toUri()
    ).build();
}
```

Figure 11 : Ressource qui permet de créer un utilisateur

« UpdateUtilisateur » est une interface implémentant l'interface « Consumer ». Cette interface a comme seule utilité d'offrir une méthode qui prend un paramètre, effectue un traitement et qui ne retourne rien. L'objectif étant de retourner une exception dans cette unique méthode dès qu'il y a un problème de validation. S'il n'y a pas de soucis, alors on fait le traitement souhaité (dans le cas présent, on crée l'utilisateur)

```
1 package fr.univlorraine.miage.revolutmiage.utilisateur.domain.cmd.updateutilisateur;
2
3 import java.util.function.Consumer;
4
5 public interface UpdateUtilisateur extends Consumer<UpdateUtilisateurInput> {
6 }
```

Figure 12 : Consumer de l'update d'un Utilisateur

```

1 package fr.univlorraine.miage.revolutmiage.utilisateur.domain.cmd.updateutilisateur;
2
3 import ...
4
5
6
7
8
9
10
11 @Service
12 @RequiredArgsConstructor
13 public class UpdateUtilisateurImpl implements UpdateUtilisateur {
14     private final UpdateUtilisateurValidator validator;
15     private final UtilisateurCatalog catalog;
16     private final PasswordEncoder passwordEncoder;
17
18     @Override
19     public void accept(final UpdateUtilisateurInput input) {
20         validator.validate(input);
21
22         final Utilisateur toSave = new Utilisateur()
23             .setNumeroPasseport(input.getNumeroPasseport())
24             .setNom(input.getNom())
25             .setPrenom(input.getPrenom())
26             .setNumeroTelephone(input.getNumeroTelephone())
27             .setPays(input.getPays())
28             .setDateDeNaissance(LocalDate.parse(input.getDateDeNaissance()))
29             .setSecret(passwordEncoder.encode(input.getPassword()));
30
31         catalog.save(toSave);
32     }
33 }

```

Figure 13 : Implémentation du Consumer Utilisateur

Dans la Figure 13 on a bien dans un premier temps la validation sur les saisies de l'utilisateur qui est appelée. Si aucune exception est retournée alors on passe à l'exécution du traitement. Comme dit précédemment, la validation étant dans le domaine, on n'appelle pas directement le repository d'utilisateur pour le sauvegarder mais bien le catalogue.

La validation se découpe en deux parties. La première concerne la validation du format des données transmises et la deuxième partie concerne plutôt la validation de la cohérence des données saisies.

Pour ce qui est de la première partie, j'utilise le moyen vu en cours qui permet de définir certaines propriétés sur le format d'un attribut.

```

UtilisateurInput.java
1 package fr.univlorraine.miage.revolutmiage.utilisateur.domain.cmd;
2
3 import ...
4
5 @Getter
6 @Setter
7 @NoArgsConstructor
8 @Accessors(chain = true)
9
10 public class UtilisateurInput {
11     @NotBlank
12     @Length(min = 1, max = 20)
13     private String nom;
14
15     @NotBlank
16     @Length(min = 1, max = 20)
17     private String prenom;
18
19     @NotBlank
20     @Pattern(regexp = "\\d{4}-\\d{2}-\\d{2}$")
21     private String dateDeNaissance;
22
23     @NotBlank
24     @Length(min = 1, max = 20)
25     private String pays;
26
27     @NotBlank
28     @Pattern(regexp = "[0-9]{2}[a-zA-Z]{2}[0-9]{5}$")
29     private String numeroPasseport;
30
31     @NotBlank
32     @Pattern(regexp = "\\+[1-9]{1}[0-9]{3,14}$")
33     private String numeroTelephone;
34 }

```

Figure 14 : Contraintes de validation pour les champs d'un utilisateur

La validation est ensuite faite dans une classe abstraite qui est identique à chaque entité. Le but étant de factoriser un maximum le code pour en réduire la dette technologique. Cette classe fait l'appel à la méthode qui permet la validation. La première validation touche le format des données. Si le format est incorrect alors on ne passe pas à la suite et on retourne une exception contenant les attributs avec leurs problèmes détectés. Si le format des données est correct on passe à la deuxième partie de la validation qui est une méthode qui sera définie dans la classe qui héritera de notre classe abstraite

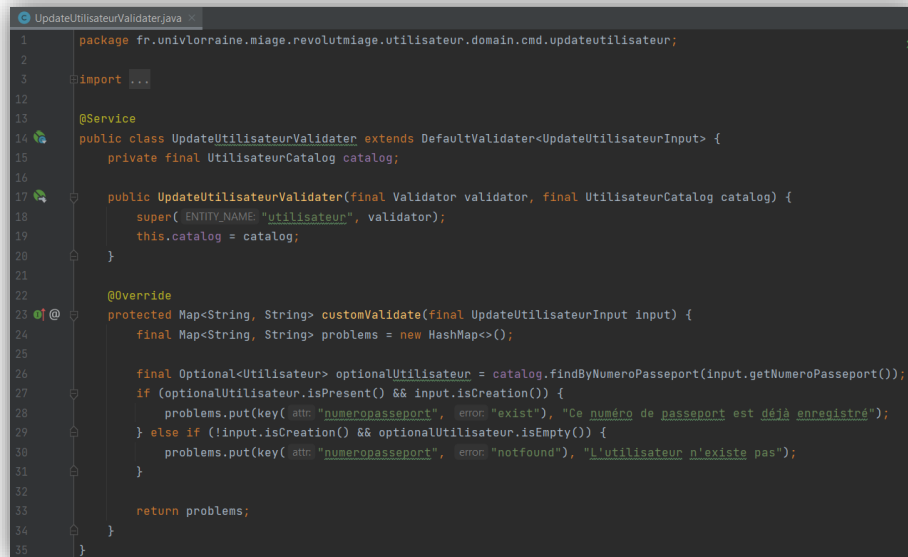
```

DefaultValidator.java
1 package fr.univlorraine.miage.revolutmiage.utils.domain.cmd;
2
3 import ...
4
5 @Service
6 @AllArgsConstructor
7
8 public abstract class DefaultValidator<INPUT> {
9     protected String ENTITY_NAME;
10     protected final Validator validator;
11
12     public void validate(final INPUT input) {
13         final Set<ConstraintViolation<INPUT>> violations = validator.validate(input);
14         if (!violations.isEmpty()) {
15             throw new ConstraintViolationException(ENTITY_NAME, violations);
16         }
17
18         final Map<String, String> problems = customValidate(input);
19         if (!problems.isEmpty()) {
20             throw new InputValidationException(problems);
21         }
22     }
23
24     protected abstract Map<String, String> customValidate(final INPUT input);
25
26     public String key(final String attr, final String error) {
27         return new StringJoiner(" ").add(ENTITY_NAME).add("validation").add(attr).add(error).toString();
28     }
29 }

```

Figure 15 : Classe abstraite qui gère la validation

Dans la classe qui hérite de « DefaultValidator » j'ai pu définir différents comportements suivant les contraintes des entités bloquer une éventuelle mauvaise saisie (impossible de créer deux utilisateurs avec le même passeport, opération sur un même compte, ...). L'utilisateur saisissant les données sera alors informé du problème lié à sa saisie.



```
1 package fr.univlorraine.miage.revolutmiage.utilisateur.domain.cmd.updateutilisateur;
2
3 import ...
4
5 @Service
6 public class UpdateUtilisateurValidator extends DefaultValidator<UpdateUtilisateurInput> {
7     private final UtilisateurCatalog catalog;
8
9     public UpdateUtilisateurValidator(final Validator validator, final UtilisateurCatalog catalog) {
10         super(ENTITY_NAME, "utilisateur", validator);
11         this.catalog = catalog;
12     }
13
14     @Override
15     protected Map<String, String> customValidate(final UpdateUtilisateurInput input) {
16         final Map<String, String> problems = new HashMap<>();
17
18         final Optional<Utilisateur> optionalUtilisateur = catalog.findByNumeroPasseport(input.getNumeroPasseport());
19         if (optionalUtilisateur.isPresent() && input.isCreation()) {
20             problems.put(key( attr: "numeropasseport", error: "exist"), "Ce numéro de passeport est déjà enregistré");
21         } else if (!input.isCreation() && optionalUtilisateur.isEmpty()) {
22             problems.put(key( attr: "numeropasseport", error: "notfound"), "L'utilisateur n'existe pas");
23         }
24
25         return problems;
26     }
27 }
```

Figure 16 : Classe qui redéfinit la méthode de validation spécifique

Il me suffit alors de faire des conditions pour vérifier la cohérence des saisies. Dans la Figure 16, si je suis entrain de créer l'utilisateur, alors je vérifie si le numéro de passeport renseigné n'existe pas déjà. Si c'est le cas alors j'ajoute une description du problème dans la Map pour pouvoir la renvoyer à l'utilisateur. Le but étant de donner un identifiant unique au problème pour qu'avec un éventuelle frontend de l'application, on puisse mettre des messages personnalisés à l'utilisateur suivant le souci rencontré. C'est à la ligne 27 de la Figure 15 que je retourne ou non une exception si jamais il y a une incohérence dans les informations saisies.

La génération d'une exception n'est pas suffisante pour renvoyer les bonnes informations à l'utilisateur. Il est nécessaire de « catcher » l'exception quand il y en a une pour faire un traitement dessus afin d'organiser les données avant de les envoyer au client. Pour se faire, Spring voit une nouvelle fois à notre secours. Il existe une annotation qui fait appel à une méthode spécifique si jamais une exception définie est levée. Il suffit alors de faire le traitement souhaité pour l'exception afin de le retourner à l'utilisateur.

```

1 package fr.univlorraine.miage.revolutmiage.utils.infra.rest;
2
3 import ...
4
12
13 public abstract class DefaultResource {
14
15     @ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY)
16     @ExceptionHandler({ConstraintViolationException.class})
17     @ExceptionHandler({InputValidationException.class})
18     public Map<String, Map<String, String>> handleValidationExceptions(final ConstraintViolationException ex) {
19         final Map<String, String> errors = new HashMap<>();
20         ex.getConstraintViolations().forEach(error -> {
21             final StringJoiner fieldName = new StringJoiner(" ");
22             fieldName.add(ex.getMessage());
23             fieldName.add("validation");
24             fieldName.add(error.getPropertyPath().toString());
25             fieldName.add(error.getConstraintDescriptor().getAnnotation().annotationType().getSimpleName());
26             final String errorMessage = error.getMessage();
27             errors.put(fieldName.toString().toLowerCase(), errorMessage);
28         });
29         return new HashMap<>() {{
30             put("problems", errors);
31         }};
32     }
33
34     @ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY)
35     @ExceptionHandler({InputValidationException.class})
36     public Map<String, Map<String, String>> handleValidationExceptions(final InputValidationException ex) {
37         return new HashMap<>() {{
38             put("problems", ex.getProblems());
39         }};
40     }
41 }

```

Figure 17 : Classe qui « catch » les exceptions pour préparer l’affichage des soucis

POST (baseUrl)/api/utilisateurs...

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```

1 {
2   "nom": "Dujardin",
3   "prenom": "Jean",
4   "dateDeNaissance": "172-06-19",
5   "pays": "France",
6   "numeroPasseport": "666RT25367",
7   "numeroTelephone": "+33606060606",
8   "password": ""
9 }

```

Body Cookies Headers (10) Test Results

Status: 422 Unprocessable Entity (WebDAV) (RFC 4918) Time: 35 ms Size: 737 B Save Response

Pretty Raw Preview Visualize JSON

```

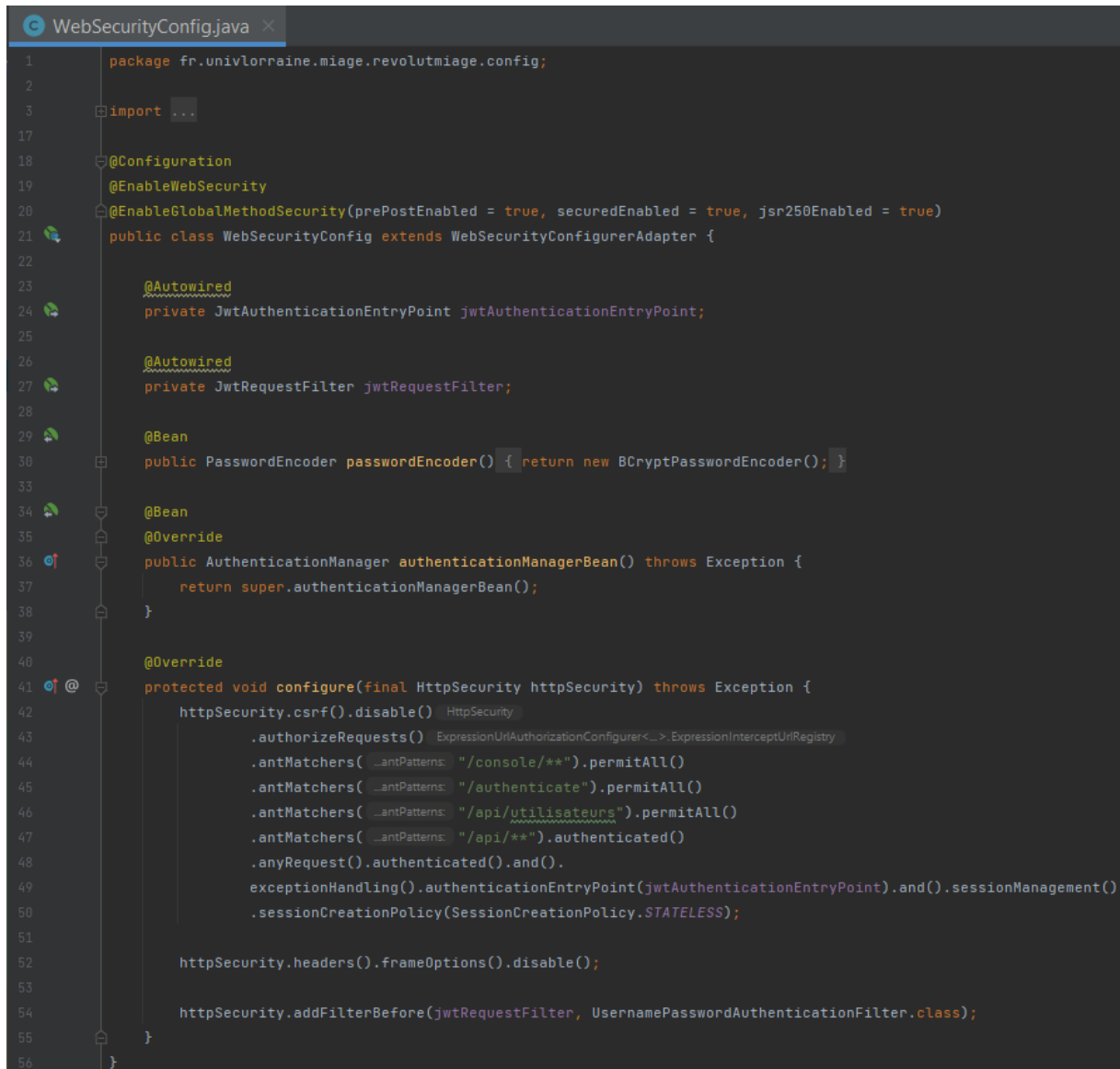
1 {
2   "problems": {
3     "utilisateur.validation.password.notblank": "ne doit pas être vide",
4     "utilisateur.validation.numeropasseport.pattern": "doit correspondre à \\^[0-9]{2}[a-zA-Z]{2}[0-9]{5}$\\",
5     "utilisateur.validation.password.length": "la longueur doit être comprise entre 5 et 50",
6     "utilisateur.validation.datedenaissance.pattern": "doit correspondre à \\^[0-9]{4}-[0-9]{2}-[0-9]{2}$\\",
7   }
8 }

```

Figure 18 : Requête avec Postman contenant des problèmes de saisie

f) La sécurité

Concernant la sécurité, Spring Security m'a permis de gagner du temps. Cette librairie rajoute une couche de sécurité concernant l'accès à l'application. Par défaut, ça bloque tous les accès et il est nécessaire de définir une politique route par route pour gérer l'accès à ceux-ci.



```
1 package fr.univlorraine.miage.revolutmiage.config;
2
3 import ...
17
18 @Configuration
19 @EnableWebSecurity
20 @EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)
21 public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
22
23     @Autowired
24     private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;
25
26     @Autowired
27     private JwtRequestFilter jwtRequestFilter;
28
29     @Bean
30     public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
31
32
33
34     @Bean
35     @Override
36     public AuthenticationManager authenticationManagerBean() throws Exception {
37         return super.authenticationManagerBean();
38     }
39
40     @Override
41     protected void configure(final HttpSecurity httpSecurity) throws Exception {
42         httpSecurity.csrf().disable().headers().frameOptions().disable();
43         .authorizeRequests().antMatchers("/console/**").permitAll()
44         .antMatchers("/authenticate").permitAll()
45         .antMatchers("/api/utilisateurs").permitAll()
46         .antMatchers("/api/**").authenticated()
47         .anyRequest().authenticated().and().sessionManagement()
48         .exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint).and().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
49
50         httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);
51     }
52
53
54
55 }
```

Figure 19 : Classe de configuration de spring-security

Par défaut, j'autorise la route pour créer un utilisateur ainsi que celle pour se connecter. Cela veut dire qu'il n'est pas nécessaire d'être connecté pour faire une requête sur ces deux routes là. La ligne 47 permet de dire que pour toutes les routes qui commence par « /api » il est nécessaire d'être authentifié sinon, l'application retourne une 401 pour indiquer que l'utilisateur n'est pas connecté. Avec une annotation, je peux ensuite définir le

rôle qui a accès à une route pour par exemple définir des routes réservées aux administrateurs ou autre.

Pour ce qui est du moyen de s'authentifier, c'est avec un numéro de passeport et un mot de passe que l'utilisateur se connecte en faisant appel à la route « /authenticate ». Si tout est bon, l'application retourne un token JWT à l'utilisateur. Il lui suffit ensuite pour chaque route, d'ajouter ce token dans le header. Initialement, je voulais utiliser l'Identity Provider Keycloak pour laisser la gestion de l'authentification à ce service mais malheureusement, je n'ai pas trouvé de solution pour créer un utilisateur sans passer par l'interface graphique (jusqu'à ce qu'un autre étudiant me la donne...) et j'ai dû gérer directement dans mon application l'authentification. Récupérer le token, vérifier s'il est valide et récupérer les informations à l'intérieur pour connaître l'identité de l'utilisateur qui essaye d'accéder à l'API (et ça n'a pas été simple à mettre en place !).

g) Les tests

Dans l'application, j'ai essayé de faire un maximum de tests pour garantir un code fonctionnel et sûr. J'étais déjà convaincu avant et je trouve que c'est une véritable sécurité de faire cela. Même si cela m'a pris beaucoup de temps, j'ai testé les mappeurs pour garantir leurs bons fonctionnements (je ne suis pas convaincu que ce soit nécessaire mais dans le doute, je l'ai fait). J'ai également testé l'API en simulant des requêtes avec différentes saisies pour garantir la couverture de tous les cas possibles. J'ai également testé les validateurs des entités pour garantir que je n'ai pas oublié un cas spécifique.

h) Hateoas

J'ai ajouté Hateoas au projet pour que les utilisateurs de l'API puissent facilement retrouver les requêtes qui sont possibles de faire. Les liens apparaissent uniquement dans des requêtes de type GET car les autres n'ont pas d'intérêt.

i) Application qui communique avec la banque

J'ai ajouté une application sur un autre port qui correspond à un paiement avec un terminal dédié. L'objectif étant de donner une carte bleue avec un code ainsi que le montant à payer. Une fois cela fait, le terminal fait une requête à notre banque application pour valider et effectuer l'opération. Je l'ai représenté de façon simple : le numéro de la carte, le code ou le sans contact, le montant sont les seules infos à transmettre quand on fait une requête avec l'API des ventes.

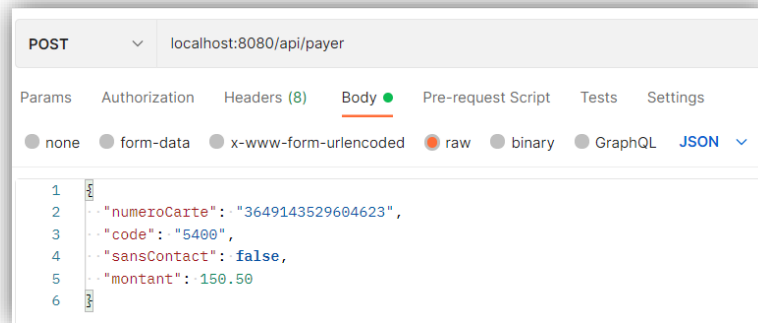


Figure 20 : Appel de l'API des ventes pour faire une opération

Cette route n'est pas sécurisée par un token car elle doit permettre à des commerces de s'y connecter. Il serait judicieux de leur fournir un token pour des raisons de sécurité mais je n'ai pas eu le temps de le faire. La validation est toujours présente et permet s'il y a le moindre souci avec le paiement de donner l'information au commerce de la même manière que le fait la banque API avec ces utilisateurs. Concernant le taux, il est fait dans la banque API pour le moment mais aurait pu être externalisé dans un service à part. Pour donner la possibilité d'ajouter des nouveaux taux de conversions, l'entité a été ajoutée en base de données pour qu'il ne soit pas nécessaire de modifier le code source pour ajouter des éléments.

j) Loadbalancing

Je n'ai pas eu le temps d'intégrer cette fonctionnalité dans mon projet. J'ai eu des soucis avec Consul et je n'ai jamais réussi à faire communiquer mes applications ensemble. Par manque de temps, je n'ai pas été plus loin malheureusement.

k) Dockerisation

Pour pouvoir créer plusieurs instances d'une application, il faut conteneuriser celles-ci pour pouvoir en redéployer d'autres. C'est pour cela que les deux applications le sont même si la réplication sera effective seulement sur la banque application.

3) Conclusion

C'était un projet vraiment top, merci beaucoup à vous. Même si c'est un des plus long de cette année, j'y ai trouvé un réel intérêt et j'ai vraiment pris plaisir. Je n'ai pas fait tout ce qui était demandé par manque de temps mais je compte mettre en place ces technologies plus tard.