Spécifications – Gestion de Comptes Utilisateur. Architecture reposant sur : Maven, l'écosystème Spring[1], AOP, MongoDB, Lombok, REST API

Spécifications Techniques : Développement de **Services REST** de **"Gestion de Comptes Utilisateur"** avec comme fonctionnalités de base[2] :

- Créer un compte utilisateur
- Afficher les détails d'un l'utilisateur existant dans le SI[3].

Version	Date	Auteur	Objet de la version
1.0	12/02/2021	VOT	Initialisation du document – Conception architecture du document
		VOT	Introduction et des exigences
		VOT	Architecture applicative et technique de l'application
		VOT	Fonctionnement global de l'application
		VOT	Aspects techniques sous-jacents
	13/02/2021	VOT	Modèles de données
		VOT	Tests et couverture de codes
	16/01/20021	VOT	Relecture et divers ajustements dans le document
	19/02/2021	VOT	Relecture/Ajout nouvelles fonctionnalités et documentation API avec Swagger

[1] Le principal élément de l'écosystème Spring utilisé pour l'intégration des composants applicatifs est : **Spring Boot**.
[2] D'autres fonctionnalités supplémentaires pourraient être proposées
[3] **SI** → **S**ystème d'**I**nformations

Sommaire

1. Introduction 3

1.1. Objectif du document 3

1.2. Le formalisme 3

2. Les exigences 3

2.1. Les exigences fonctionnelles de base 4

2.2. Les nouvelles fonctionnalités proposées 4

2.3. Les exigences non fonctionnelles du socle applicatif 5

3. Architecture technique et applicative 5

3.1. Choix de l'architecture du socle applicatif 5

3.2. Le diagramme d'architecture applicative et technique 6

4. Fonctionnement global 8

4.1. Le diagramme des cas d'utilisation 8

4.2. Le processus de création d'un nouveau compte utilisateur 9

4.3. Le processus d'affichage des données d'un compte utilisateur 10

5. Aspects techniques sous-jacents 11

5.1. Le langage 11

5.2. La structure applicative 11

5.3. Organisation du code source 11

5.4. La gestion du cycle de vie des objets 11

5.5. La gestion de la persistance des données et Transactions 12

5.6. La gestion des logs applicatifs 12

5.7. La gestion des erreurs/exceptions 12

5.8. Environnement d'exécution 13

5.9. Swagger UI /OpenAPI 13

5.10. Le code source de l'application et documentation 14

6. Le modèle de données 14

7. Les Tests et Couverture de codes 15

7.1. Les types de tests 15

7.2. Les outils de tests 15

7.1. Les rapports de couverture de codes 16

1. Introduction

Le sujet qui fait l'objet de ce document est né de ma candidature au poste de **Tech Lead Java/Jee** chez le client pour son "***Labs***". Il s'agit de créer deux REST API permettant d'exposer chacune une fonctionnalité précise dans le cadre de la gestion des comptes et informations des utilisateurs dans le *SI à développer* (le **Back-End**).

Le présent document est le dossier de spécifications techniques détaillées (de façon macroscopique) du nouveau système à développer pour exposer les fonctionnalités issues des besoins exprimés.

1.1. Objectif du document

C'est de fournir une vue d'ensemble non seulement des fonctionnalités à développer, mais également de faciliter la compréhension des besoins exprimés et l'implémentation technique qui en découle.

Il n'a pas pour objectif de présenter en détails les grandes fonctionnalités, mais plutôt de présenter les environnements, les objets, les interactions entre composants, …, qui concourent à la bonne réalisation de ce besoin, puis au bon fonctionnement de l'application. Ainsi, il permet de décrire les éléments de l'architecture du socle applicatif et également fournir une vision globale des flux d'échanges entre l'application et les différents acteurs et/ou briques/composants applicatifs selon les aspects suivants entre autres :

- Les exigences
- Les éléments architecturaux du socle applicatif
- Aspects techniques sous-jacents
- Le modèle de données métier
- Les Tests, le packaging et livrables

1.2. Le formalisme

Dans la suite de ce document, le formalisme choisi pour la modélisation est **UML**. Ainsi, il permettra de fournir une série de diagrammes permettant de faciliter la compréhension du besoin lors de sa réalisation. Les différents diagrammes et/ou schémas fournis, sont réalisés avec **EA**[4].

2. Les exigences

Le principal besoin exprimé est de créer deux REST Services (API), permettant d'exposer des fonctionnalités. De cette expression de besoin, il en ressort deux grandes catégories d'exigences. Ce sont donc :

[4] **EA** : **E**nterprise **A**rchitect → outil complet d'analyse et de conception d'UML, un outil graphique conçu pour aider à établir un logiciel robuste et maintenable.

- Les exigences fonctionnelles
- Les exigences non fonctionnelles

2.1. Les exigences fonctionnelles de base

Elles relèvent de l'aspect métier donc des fonctionnalités exposées ou proposées par l'application à développer. La lecture de l'expression des besoins permet d'identifier les deux grands processus cités ci-dessous qui composent les exigences fonctionnelles. On a donc :

- Le processus de gestion de la **création du compte utilisateur** dans le SI
- Le processus de gestion de la **remontée (pour affichage) des données** d'un utilisateur existant

A ceci s'ajoute **les règles métiers suivantes** :

- Valider les entrées (les données saisies par l'utilisateur ou le client)
- Seuls les personnes adultes (+18 ans) et habitant la France sont autorisées à créer un compte

Le tableau ci-dessous fournit un résumé des éléments constitutifs de chaque processus identifié de l'expression des besoins. On a donc :

Processus	Fonctionnalités
Gestion de la création du compte utilisateur	- Récupérer les données en entrée, - Vérifier/Valider les données en entrée (adulte français donc + 18ans), - Vérifier que l'utilisateur n'existe pas déjà en base de données, - Persister de nouvelles données utilisateur dans la base de données, - Gérer les exceptions/erreurs.
Gestion de la remontée des données d'un utilisateur existant	- Récupérer le critère de recherche, - Rechercher existence données utilisateur en base de données, - Remonter les données utilisateur correspondant au critère de recherche, - Gérer les exceptions/erreurs

2.2. Les nouvelles fonctionnalités proposées

De ce qui précède afin de disposer d'un système d'informations complet au sens CRUD[5] par rapport aux fonctionnalités de base de l'expression de besoins, de nouvelles fonctionnelles sont proposées. Ainsi, les fonctionnalités de base présentées ci-dessus seront complétées par celle citées ci-dessous :

- La gestion de la **mise à jour des données** d'un utilisateur existant dans le SI
- La gestion de la **suppression des données** d'un utilisateur existant dans le système
- **Afficher les détails des utilisateurs** du système (par **pagination**)

[5] **CRUD** : **C**reate (Créer) **R**ead (Lire/Récupérer) **U**pdate (Mettre à jour) **D**elete (Supprimer)

Le tableau ci-dessous fournit un résumé des nouvelles fonctionnalités en complément.

Processus	Fonctionnalités
Gestion de la mise à jour compte utilisateur	Les fonctionnalités sont identiques à celle de la gestion de la création d'un compte à la seule différence que l'utilisateur à mettre à jour existe déjà dans le système d'informations
Gestion de la suppression des données du compte d'un utilisateur existant	➢ Rechercher existence données utilisateur en base de données, ➢ Supprimer les données utilisateur correspondant au critère de recherche, ➢ Gérer les exceptions/erreurs
Gestion de l'affichage de l'ensemble des utilisateurs du système	➢ Définir les critères de pagination des données à remonter ➢ Rechercher les données utilisateur du SI ➢ Afficher les données de l'ensemble ➢ Gérer les exceptions/erreurs

2.3. Les exigences non fonctionnelles du socle applicatif

Celles-ci ont un caractère technique, mais concourent au bon fonctionnement de l'application. Selon les prérequis, on peut donc citer grosso modo:

- La gestion des accès à la base de données → donc aux informations stockées en base
- La gestion des logs applicatifs
- La gestion des exceptions/erreurs
- La gestion de configurations et de propriétés applicatives

3. Architecture technique et applicative

Il s'agit de fournir une vision globale des flux d'échanges entre l'application et les acteurs du système ou des briques/composants applicatifs.

3.1. Choix de l'architecture du socle applicatif

Dans le cadre des réalisations techniques de ce besoin, pour faire simple le choix du type de l'architecture opéré est : **architecture monolithique de type REST**[6]. Dans ce type d'architecture :

- chaque objet est représenté par une URL,

[6] **REST** → **RE**presentational **S**tate **T**ransfer

- utilisation du protocole HTTP et de ses verbes,
- le traitement de l'objet s'effectue systématiquement sans état.

3.2. Le diagramme d'architecture applicative et technique

Plus précisément ce diagramme fournit/apporte des informations sur :

- les flux d'informations entre l'application et les autres acteurs du système d'information dans lequel il va être implanté (usagers, base de données, ….)
- les flux d'informations entre les différentes pièces (briques ou composants) de l'application
- les flux entre les différents appareils du système d'informations (serveur, ordinateur, …)

Le diagramme d'architecture est donc celui présenté ci-dessous et comporte les éléments suivants :

- le **Back-End** → qui embarque les composants applicatifs permettant d'implémenter le métier
- la base de données **NoSQL** (**MongoDB**) → pour la persistance des données utilisateur
- le **client** → qui consomme les fonctionnalités exposées par les deux services REST.

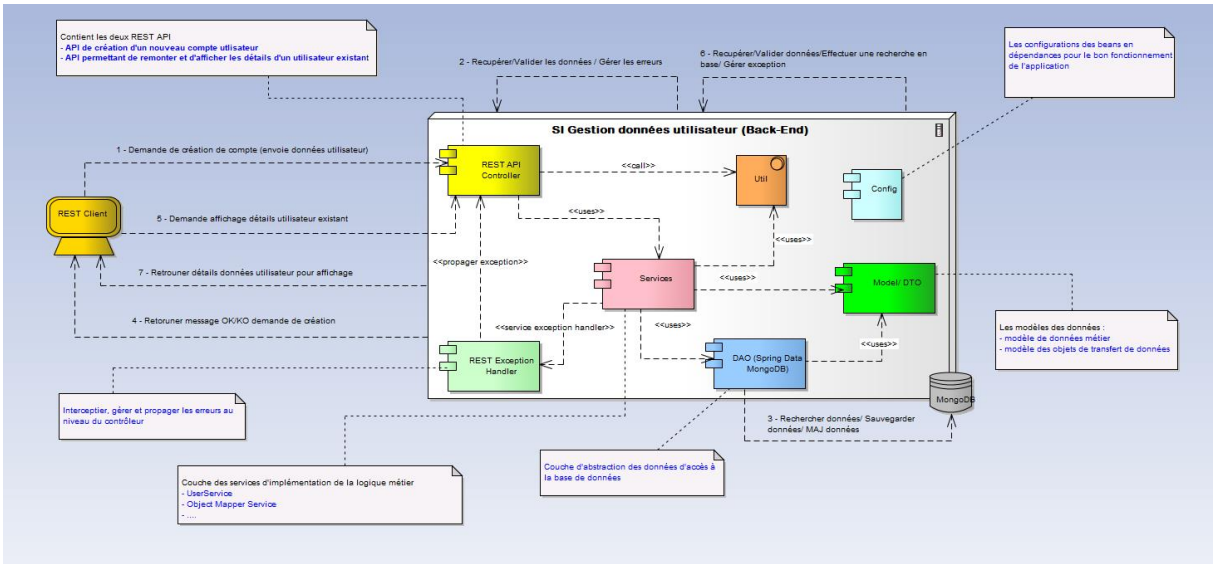


Figure 1 : Diagramme d'architecture technique et applicative

Ici, le "**Client**" peut être un utilisateur ou tout autre système informatique ou non, pouvant échanger avec le back-end qui est l'élément qui fait l'objet de ce dossier de spécifications techniques détaillées.

Le tableau ci-dessous présente la description des flux du diagramme d'architecture présenté ci-dessus. On a donc :

N°	De	Vers	Protocole utilisé ou nature
1	CLIENT	SERVER	HTTP/HTTPS
2	SERVER	SERVER	LOCALHOST
3	SERVER	MONGODB	JDBC (MongoDB)
4	SERVER	CLIENT	HTTP/HTTPS
5	CLIENT	SERVER	HTTP/HTTPS
6	SERVER	SERVER	LOCALHOST
7	SERVER	CLIENT	HTTP/HTTPS

Une vision macroscopique de la cinématique ou workflow des informations dans le schéma d'architecture ci-dessus(les composants applicatifs) est fournie par le diagramme de séquences ci-dessous.

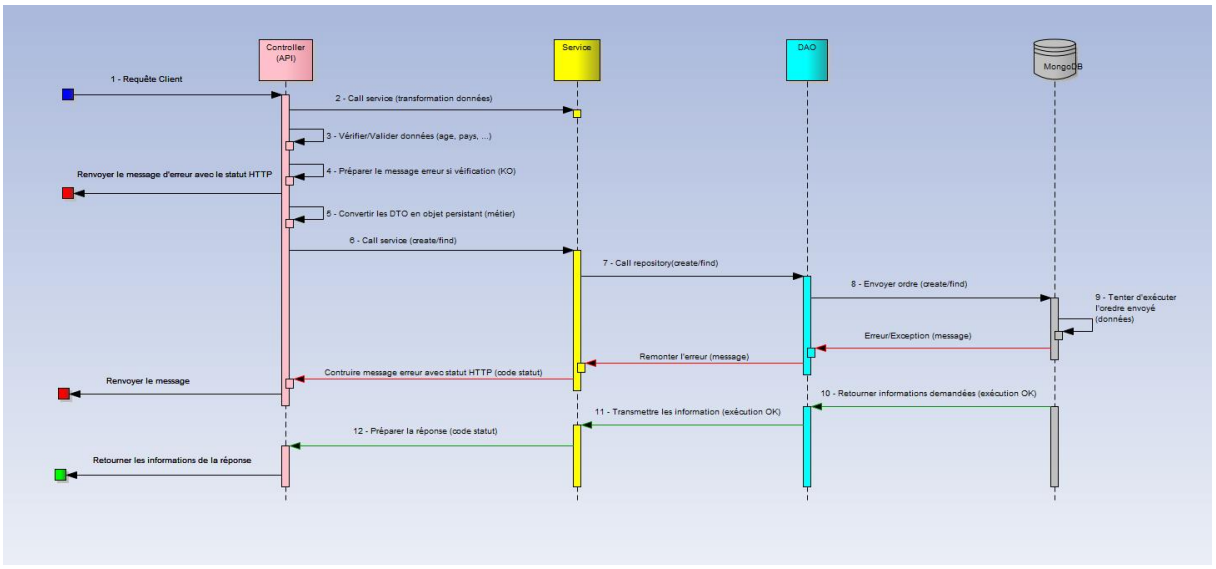


Figure 2 : Workflow des informations dans l'architecture applicative

4. Fonctionnement global

Dans cette section, sont brièvement présentés quelques éléments du fonctionnement de l'application au travers de diagrammes UML

4.1. Le diagramme des cas d'utilisation

D'un point de vue macroscopique le schéma global des cas d'utilisation est fourni par le diagramme ci-dessous. Dans ce diagramme, on y trouve l'ensemble des exigences fonctionnelles ci-dessous cités :

- Les exigences fonctionnelles de base
- Les nouvelles fonctionnalités → fournie sous forme de package

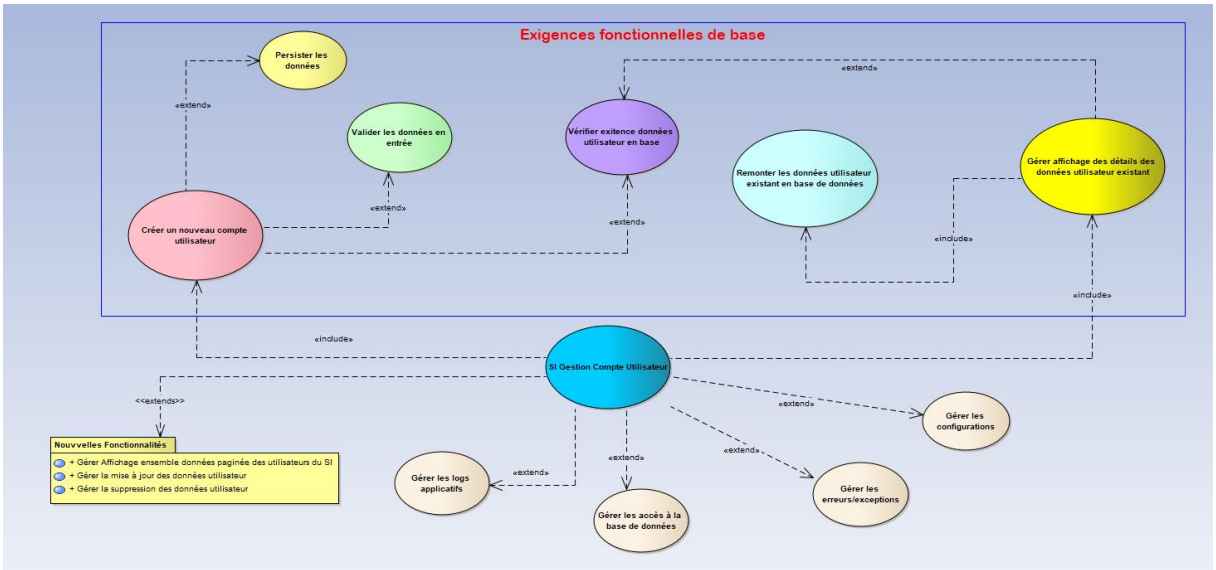


Figure 3 : Digramme global des cas d'utilisation

Attention

Dans ce qui va suivre, plus précisément au niveau des diagrammes de séquences fournis les bouts d'URL d'accès aux ressources mentionnés sont fournis à titre d'exemple.

Exemple :

- pour le *POST* : /api/user/register
- pour le *GET* : /api/user/{id}

Lors de la réalisation, dans le code source **ceux-ci peuvent être nommés autrement** du moment où ceci a un sens et est bien parlant et se rattache bien à ce qu'on veut faire ou ce qui est attendu.

Afin de ne pas alourdir les spécifications, par la suite il ne sera présenté que les processus liés aux fonctionnalités de base (car les nouvelles fonctionnalités s'y retrouvent à peu de choses près).

4.2. Le processus de création d'un nouveau compte utilisateur

Une vue macroscopique du fonctionnement global de ce processus dans l'application est fournie par le diagramme de séquences ci-dessous.

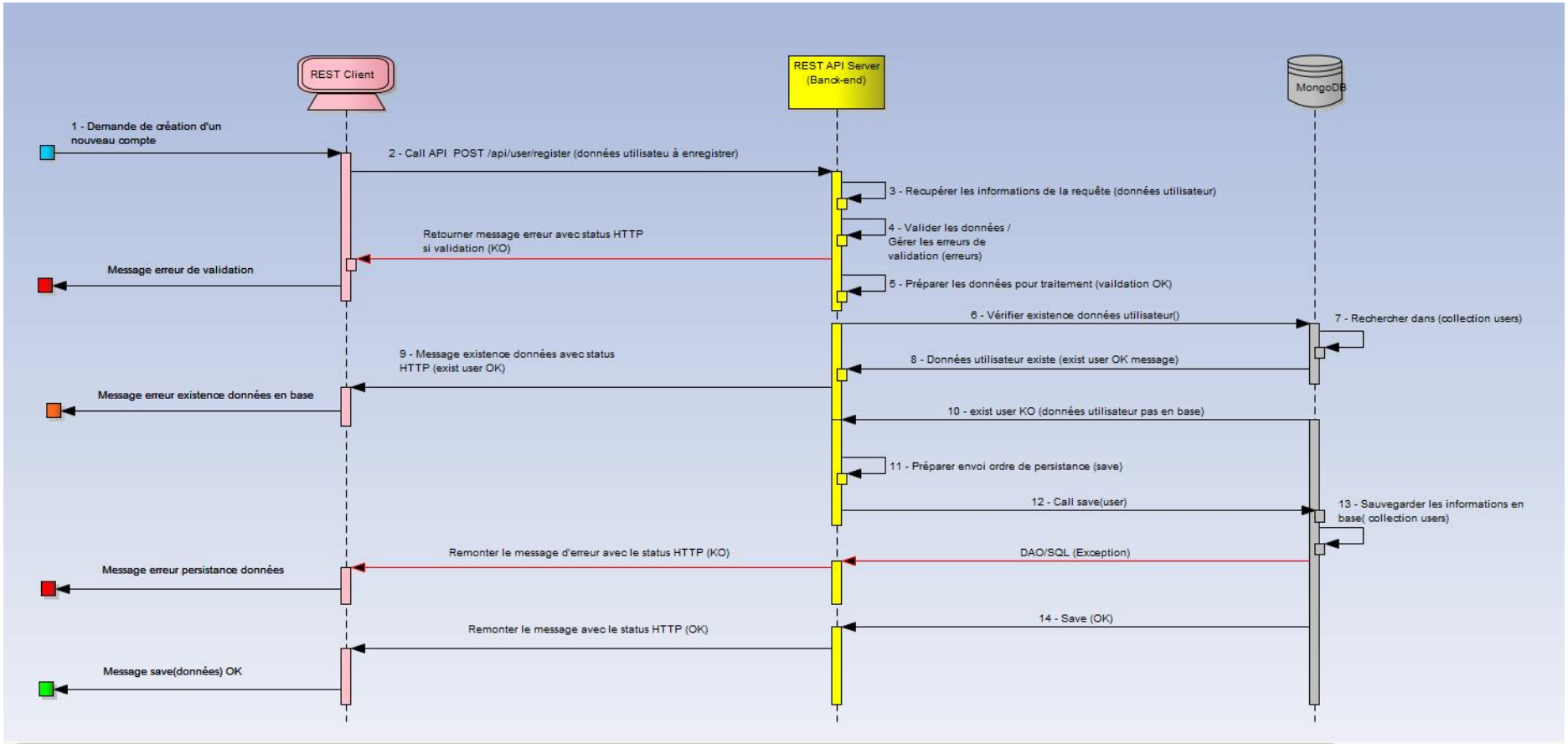


Figure 4 : Diagramme de Séquences ajout d'un nouveau compte utilisateur

4.3. Le processus d'affichage des données d'un compte utilisateur

Une vue macroscopique du fonctionnement global de ce processus dans l'application est fournie par le diagramme de séquences ci-dessous.

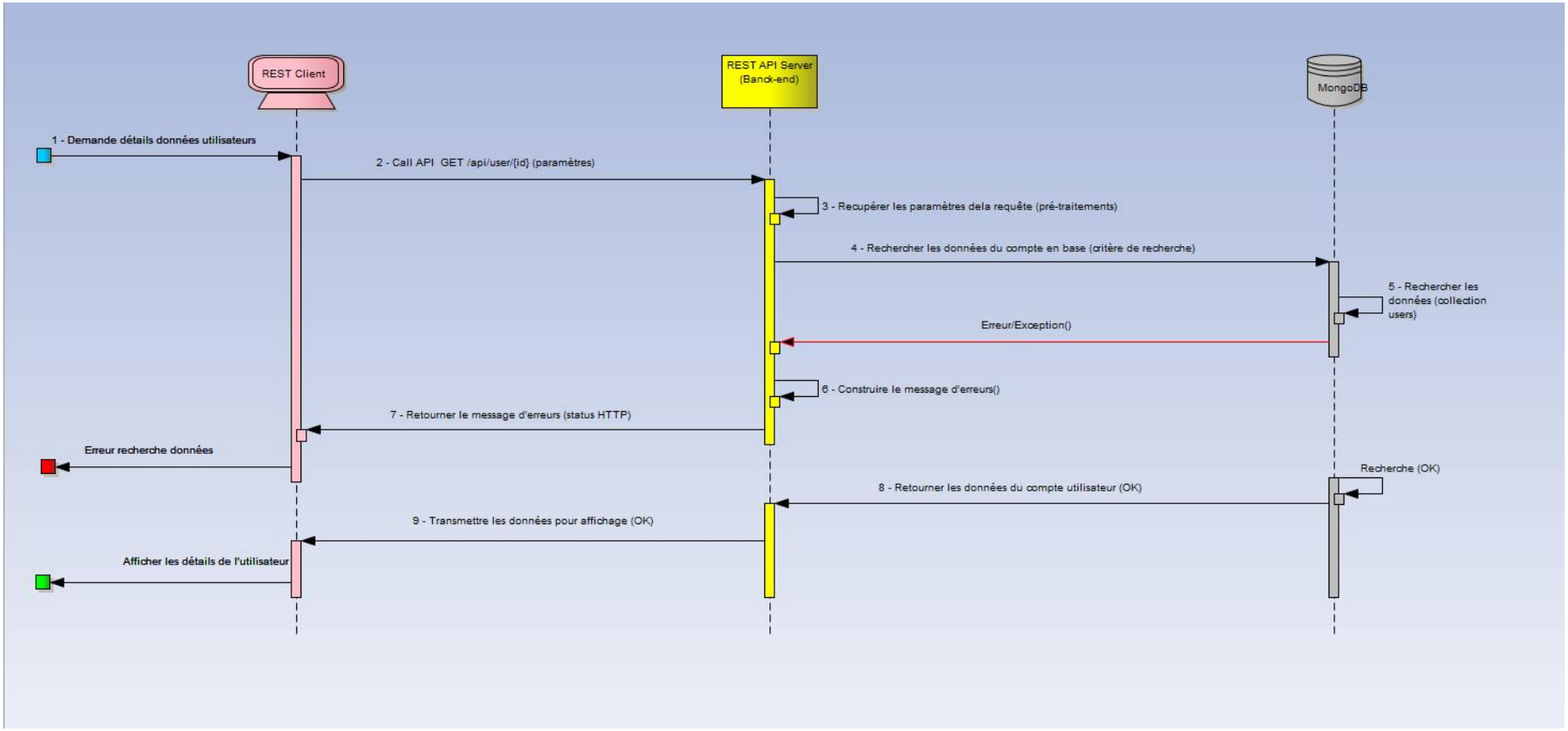


Figure 5 : Diagramme de Séquences Afficher les détails de l'utilisateur

5. Aspects techniques sous-jacents

Dans le cadre de cette réalisation, l'**utilisation des annotations** est privilégiée par rapport aux fichiers de configurations classiques XML, hormis les fichiers de configuration des logs dans l'application.

5.1. Le langage

Java dans sa version 8 est le langage de programmation de base choisi pour l'implémentation des besoins.

5.2. La structure applicative

Maven est l'outil utilisé pour gérer la construction du cycle de vie du projet Java que constitue l'application à développer. Ainsi la structure applicative du code source sera conforme aux spécifications de l'outil Maven.

5.3. Organisation du code source

Le code source de l'application sera organisé suivant la structure applicative présentée ci-dessus et par couches à savoir à minima les packages ci-dessous :

- model → contiendra les objets aussi bien persistants que non (par exemple les DTO):
- repository :→ qui contient les objets de la couche d'accès à la base de données (DAO).
- services → contient les objets embarquant le code métier et règles de gestion dans l'application
- config → contient les objets de configurations de l'application.

5.4. La gestion du cycle de vie des objets

Le développement et l'architecture de l'application repose sur l'utilisation de l'écosystème **Spring** avec intégration à minima des éléments ci-dessous :

- **Spring Boot** → *spring-boot-starter* pour activer le support Spring boot
- **Spring pour le Web** → *spring-boot-starter-web* pour activer le support Spring pour le Web
- **Spring pour MongoDB** → *spring-boot-starter-data-mongodb* pour activer le support

L'utilisation de l'écosystème **Spring** permet de bénéficier des divers apports tout en garantissant la gestion du cycle de vie des objets dans l'application. Comme autres apports on peut citer :

- AOP → Programmation Orientée Aspect. Il vise à accroître la modularité en permettant la séparation des préoccupations transversales. Ainsi, dans l'application permettra d'intercepter et tracer (journalisation de l'exécution):
  - les entrées et sorties des méthodes et les paramètres de celles-ci.
  - calculer et fournir la durée d'exécution d'une méthode
  - les erreurs survenues lors du fonctionnement de l'application
- Injection des dépendances → patron de conception permettant de découpler les dépendances entre les objets. Il permet de fournir automatiquement aux objets leurs dépendances au lieu de les créer ou de les recevoir en tant que paramètres en les injectant de manière **type-safe**.

- Inversion de contrôle → patron d'architecture, qui fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application, mais du Framework.

5.5. La gestion de la persistance des données et Transactions

L'objet de gestion de la persistance ou qui permettra d'effectuer les opérations sur les données en base, devra exploiter (étendra) l''interface : **MongoRepository** → fournit par la dépendance maven qui permet d'activer le support Spring pour la base MongoDB dans l'application (spring-boot-starter-data-mongodb).

La **gestion de la transaction** sera déléguée au Framework **Spring** qui dispose de tous les éléments pour y assurer la gestion. Elle sera indiquée par annotation au niveau de la couche "service".

5.6. La gestion des logs applicatifs

Il s'agit de gérer des messages émis par l'application durant son exécution et de permettre non seulement son exploitation immédiate ou a posteriori, mais pour effectuer le débogage, des analyses, des audits (de sécurité).

Ainsi, le choix d'outlil porte sur le Framework **Logback** comme implémentation (afin de bénéficier de ses nombreux avantages) et l'interface **SLF4J** comme logger. La configuration dans l'application se fera au travers d'un fichier de configuration XML : "logabck.xml" ou "logabck-spring.xml".

5.7. La gestion des erreurs/exceptions

Afin de permettre à l'application de remonter de façon très claire les différentes causes de dysfonctionnement, il sera mis en place des structures objets permettant de remonter les informations sur ces causes de dysfonctionnements aussi bien côté Java qu'au niveau REST API.

Le mécanisme de gestion des erreurs de l'application, doit pouvoir remonter les causes de dysfonctionnement avec le statut HTTP adéquat. Au final :

- **Côté Java** → la personnalisation de la classe d'interception et de remontée des erreurs hérite (étend) la classe mère RuntimeException. Pour simplifier sa définition est la suivante :

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class AppCustomException extends RuntimeException
{ … }
```

- **Côté API** → la personnalisation doit pouvoir intercepter pour simplifier trois grands à savoir :
  - les erreurs de type NOT_FOUND → pour l'exception personnalisée de l'application
  - les erreurs de type INTERNAL_SERVER_ERROR → erreur de type "Exception"
  - les autres erreurs remontées des exceptions HTTP Client

5.8. Environnement d'exécution

Au-delà des points énumérés ci-dessous la configuration de l'application doit permettre de faciliter son exécution dans n'importe environnement (interopérabilité). Ainsi, elle doit permettre à minima de :

- Exécuter dans n'importe quel EDI[7] ou en utilisant Maven
- Exécuter en lignes de commande à partir de l'archive (.jar) → ceci suppose la présence de JDK/JRE sur l'environnement d'exécution. .
- Fournir une configuration pour les accès à la base de données
- Démarrer l'application avec un jeu de données de tests prédéfinis.

Les éléments de configuration des identifications et accès à la source de données sont les suivants :

- *Credentials* (gestion de la sécurité) → les informations d'identification (**pas vraiment indispensable dans notre cas de figure**, mais ceci permet de disposer cette configuration au cos où) :

```
#  Credentials dans le cas où la connexion est sécurisée
spring.data.mongodb.authentication-database=<authetification>
spring.data.mongodb.username=<username>
spring.data.mongodb.password=X<mot_de_passe>
```

- *Propriétés de connexion à la source de données* → information de la chaîne de connexion à la base

```
# DB pour la base de test par exemple
spring.data.mongodb.database=admin
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
```

5.9. Swagger UI /OpenAPI

L'outil Swagger UI (du projet open source Swagger/OpenAPI) dans sa version 3.0, sera embarqué pour générer/produire la documentation des REST API de l'application. Il offre également une interface permettant d'explorer et tester les différentes méthodes offertes par le service. Il sera mis en place par une configuration Java au travers d'annotations pour la fourniture des "beans" Spring de production et injection des "Docket".

```
@Configuration
@EnableOpenApi
public class SwaggerConfig
{…… }
```

La mise en place nécessite à minima les dépendances maven ci-dessous :

[7] **EDI** → **E**nvironnement de **D**éveloppement **I**ntégré

```
<!-- OpenAPI/Swagger UI Spring Boot Starter -->
        <dependency>
                <groupId>io.springfox</groupId>
                <artifactId>springfox-boot-starter</artifactId>
                <version>3.0.0</version>
        </dependency>
        <dependency>
                <groupId>io.springfox</groupId>
                <artifactId>springfox-swagger-ui</artifactId>
                <version>3.0.0</version>
        </dependency>
```

5.10. Le code source de l'application et documentation

La gestion de la version du code est effectuée par **Git**. Le code source est donc disponible dans l'espace de travail **Git Hub** à l'adresse suivante : https://github.com/samson06/labs-tests-technique

La documentation **JavaDoc** du code source est obtenue au "*build*" des sources et est située dans le fichier : "index.html" sous "/target/apidocs/" ou dans **l'archive** : "*XXX-YYY-javadoc.jar*" sous le dossier : "/target/ "

6. Le modèle de données

Dans l'application, les modèles de données sont les suivants :

- Le modèle objet de données métier → les données utilisateur à persister en base de données
- Le modèle objet de transfert des données correspondantes → le DTO associé à l'objet métier.

Les champs ou attributs du modèle objet métier de gestion des données des utilisateurs dans le système d'informations étant à identifier, le tableau ci-dessous fournit une liste des attributs proposés et leur caractéristique.

Attributs	Détails	Type Java	Taille	Etat
id	Identifiant technique auto-généré (unique)	String		Non Null
firstName	Le prénom de l'utilisateur	String	80	Non Null
lastName	Le nom de famille de l'utilisateur	String	50	Non Null
email	Adresse mail de l'utilisateur (unique)	String	50	Non Null
age	L'âge de l'utilisateur	Integer		Non Null
country	Le pays de naissance de l'utilisateur (nationalité)	String	50	Non Null
adresse	Adresse du lieu de résidence de l'utilisateur	String		
city	La ville de résidence de l'utilisateur	String		
phone	Le numéro de téléphone de l'utilisateur	String		

Dans le tableau ci-dessus les attributs qui sont à "**Non**" pour la colonne "**Vide**", ont un caractère obligatoire.

Le modèle objet de transfert des données qui en résulte possède les mes mêmes attributs.

7. Les Tests et Couverture de codes

Les outils de tests classiques de **Java** et **Spring** sont utilisés pour effectuer les différents types de tests lors de la phase d'implémentation.

7.1. Les types de tests

Compte tenu du périmètre du besoin à implémenter, les types de tests qui seront réalisés sont les suivants :

- **Test Unitaires** → ils ont un caractère obligatoire, pas seulement pour un effet de test immédiat du code, mais également permettre d'effectuer des tests de non-régression lors de modifications qui interviendront inévitablement durant la vie de l'application.
- **Tests d'Intégration** → s'assurer que le comportement de l'application est toujours aussi conforme, au fur et à mesure de l'assemblage des unités de code. Ils sont de deux types : *les tests d'intégration composants* et les *tests d'intégration système*.

7.2. Les outils de tests

La partie **test** de l'écosystème **Spring** (Framework de base de l'application) plus précisément sa composante : "*spring-boot-starter-test*", (spring-test, spring-boot-test, spring-boot-test-autoconfigure), fournit des outils permettant la réalisation des types de tests cités ci-dessus.

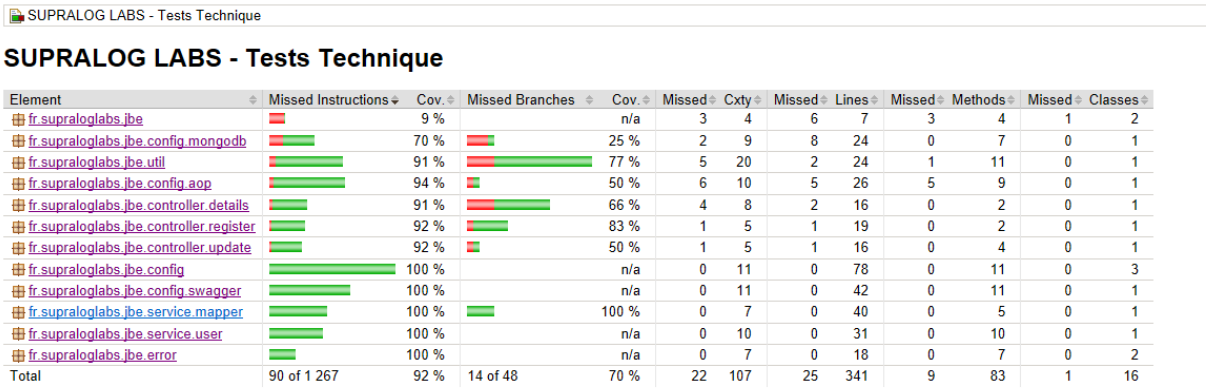
Le tableau ci-dessous fournit une liste non exhaustive des principaux outils à disposition et à utiliser pour effectuer les tests dans le cadre de cette application.

Framework	Détails
Mockito	pour les mocks
JUnit 5	pour l'écriture des classes des Tests Unitaires et d'intégration.
Assert-J	Pour les assertions de tests
Postman	pour tester les fonctionnalités exposées par les REST API
Swagger	Pour générer la documentation et Tester les REST API

7.1. Les rapports de couverture de codes

Le *plugin maven JaCoCo* sera intégré au projet pour produire les rapports de couverture de codes sur la base des classes des Tests. Il est fourni au travers du fichier : **index.html** qui se situe à l'emplacement : /target/jacoco/test/index.html.

Voici, selon la configuration qui sera mise ne place lors de la réalisation du besoin une copie d'écran des rapports fournis par l'outil.



SUPRALOG LABS - Tests Technique

SUPRALOG LABS - Tests Technique

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.supraloglabs.jbe		9 %		n/a	3	4	6	7	3	4	1	2
fr.supraloglabs.jbe.config.mongodb		70 %		25 %	2	9	8	24	0	7	0	1
fr.supraloglabs.jbe.util		91 %		77 %	5	20	2	24	1	11	0	1
fr.supraloglabs.jbe.config.aop		94 %		50 %	6	10	5	26	5	9	0	1
fr.supraloglabs.jbe.controller.details		91 %		66 %	4	8	2	16	0	2	0	1
fr.supraloglabs.jbe.controller.register		92 %		83 %	1	5	1	19	0	2	0	1
fr.supraloglabs.jbe.controller.update		92 %		50 %	1	5	1	16	0	4	0	1
fr.supraloglabs.jbe.config		100 %		n/a	0	11	0	78	0	11	0	3
fr.supraloglabs.jbe.config.swagger		100 %		n/a	0	11	0	42	0	11	0	1
fr.supraloglabs.jbe.service.mapper		100 %		100 %	0	7	0	40	0	5	0	1
fr.supraloglabs.jbe.service.user		100 %		n/a	0	10	0	31	0	10	0	1
fr.supraloglabs.jbe.error		100 %		n/a	0	7	0	18	0	7	0	2
Total	90 of 1 267	92 %	14 of 48	70 %	22	107	25	341	9	83	1	16