



# **Spécifications techniques de l'application**



Révisio n	Rédacteurs	Date	Objet
1	A. Mochizuki	01/02/2024	Création du document
2	A.Mochizuki	03/02/2024	Ajout du script de migration base de données
3	A.Mochizuki	05/02/2024	Modification du MCD
4	A.Mochizuki	21/02/2024	MAJ MCD + ajout vue couches

---

# 1 INTRODUCTION

---

## 1.1 *Objet du document*

---

Ce document a pour objectif de présenter l'essentiel des questions techniques liées à la mise en place de l'application Inote

Ce document présente :

- Le Modèle conceptuel de données
- Le modèle physique de données
- L'architecture de l'application
- Les outils choisis
- La stratégie de développement
- La stratégie de test

### Table des matières

---

## Table des matières

## Table des matières

<b>1 INTRODUCTION.....</b>	<b>3</b>
1.1 Objet du document.....	3
<b>2 SOMMAIRE.....</b>	<b>4</b>
<i>Table des matières.....</i>	<i>4</i>
<b>3 ARCHITECTURE LOGICIELLE.....</b>	<b>5</b>
3.1 Produits et versions.....	5
<b>4 FOCUS TECHNIQUE.....</b>	<b>7</b>
4.1 Organisation des données.....	7
4.2 Diagramme de classes métier (Modèle Conceptuel de données).....	7
4.3 Modèle Logique de données (mapping MCD).....	8
4.4 Script SQL de création de la base de données.....	8
4.5 Architecture 3-Tiers de application (couches).....	9
4.6 Règles de développement coté back.....	10
<b>5 TESTS ET INTÉGRATION.....</b>	<b>14</b>
5.1 Stratégie de tests unitaires.....	14
5.2 Stratégie de tests d'intégration.....	16

## 3 ARCHITECTURE LOGICIELLE

### 3.1 Produits et versions

- Langages

Nom	Version	Commentaire
Java	17	Langage de programmation côté BackEnd
TypeScript 5.3	5.3	Langages de programmation côté FRONT-END pour l'implémentation du comportement de l'application
HTML (HyperText Markup Language)	5	Langage de programmation côté FRONT-END pour l'implémentation de la structure des vues de l'interface
CSS (Cascading Style Sheets)	4	Langage de programmation côté FRONT-END pour l'implémentation de la présentation des vues de l'interface

- Framework et dépendances côté backend

Nom	Version	Commentaires
Spring Boot	3.2	Framework de gestion du projet, facilitateur pour l'utilisation de l'écosystème Spring.
Spring Security		Starter Spring Boot dédié à l'authentification et la gestion des accès
Spring Validation		Starter Spring Boot dédié à la validation des entrées utilisateur
Spring Web		Starter Spring Boot utilisé ici pour l'implémentation d'une API de type REST (permet d'embarquer également un serveur http embarqué)
Spring-boot-starter-test		Test de l'application. Inclus notamment JUnit(tests unitaires) et Mockito(doublures).
Lombok	1.18	Bibliothèque dédiée à la simplification de l'écriture des classes Java via annotations.
springdoc-openapi-starter-webmvc-ui	2.1	Interface WEB de présentation et de test de l'API
mysql-connector-java	8.0.33	Driver MySQL

<b>H2</b>	2.2	Génération de base de données de type H2 (in memory) pour le test
<b>spring-boot-starter-data-redis</b>		Starter pour la gestion de base de données NoSQL (Key-Value) Redis (utilisé pour la sécurité)

- **Framework et dépendances côté Front-end**

Nom	Version	Commentaires
<b>Angular</b>	12	Framework côté Front dédié à l'implémentation de l'interface utilisateur.
<b>Ionic ? Flutter ? autre ?</b>		Framework pour l'application mobile
<b>BootStrap</b>	5.2	Présentation de l'interface

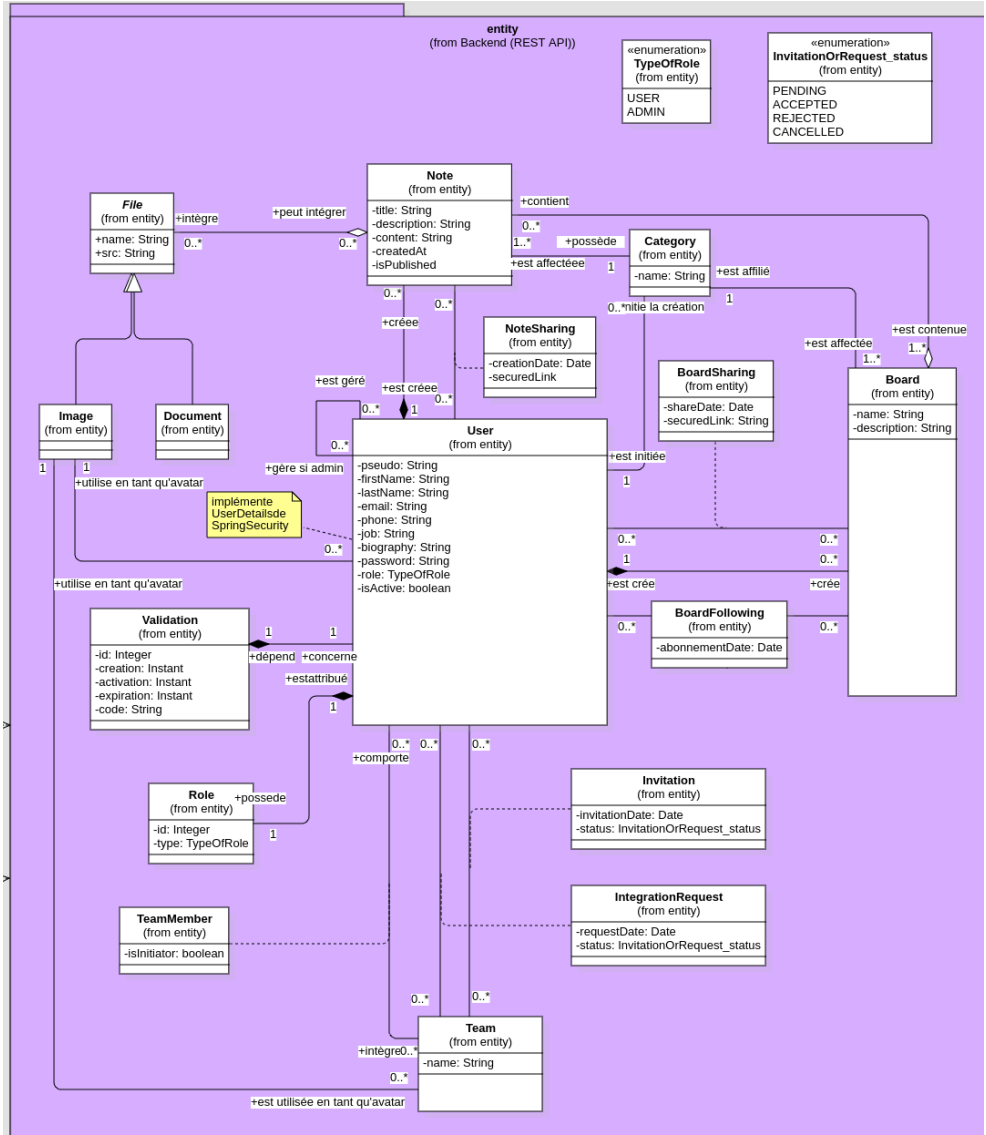
- **Serveur de base de données**

Nom	Version	Commentaires
<b>MySQL</b>	8.2	SGBDR

## 4 FOCUS TECHNIQUE

## 4.1 Organisation des données

#### 4.2 Diagramme de classes métier (Modèle Conceptuel de données)

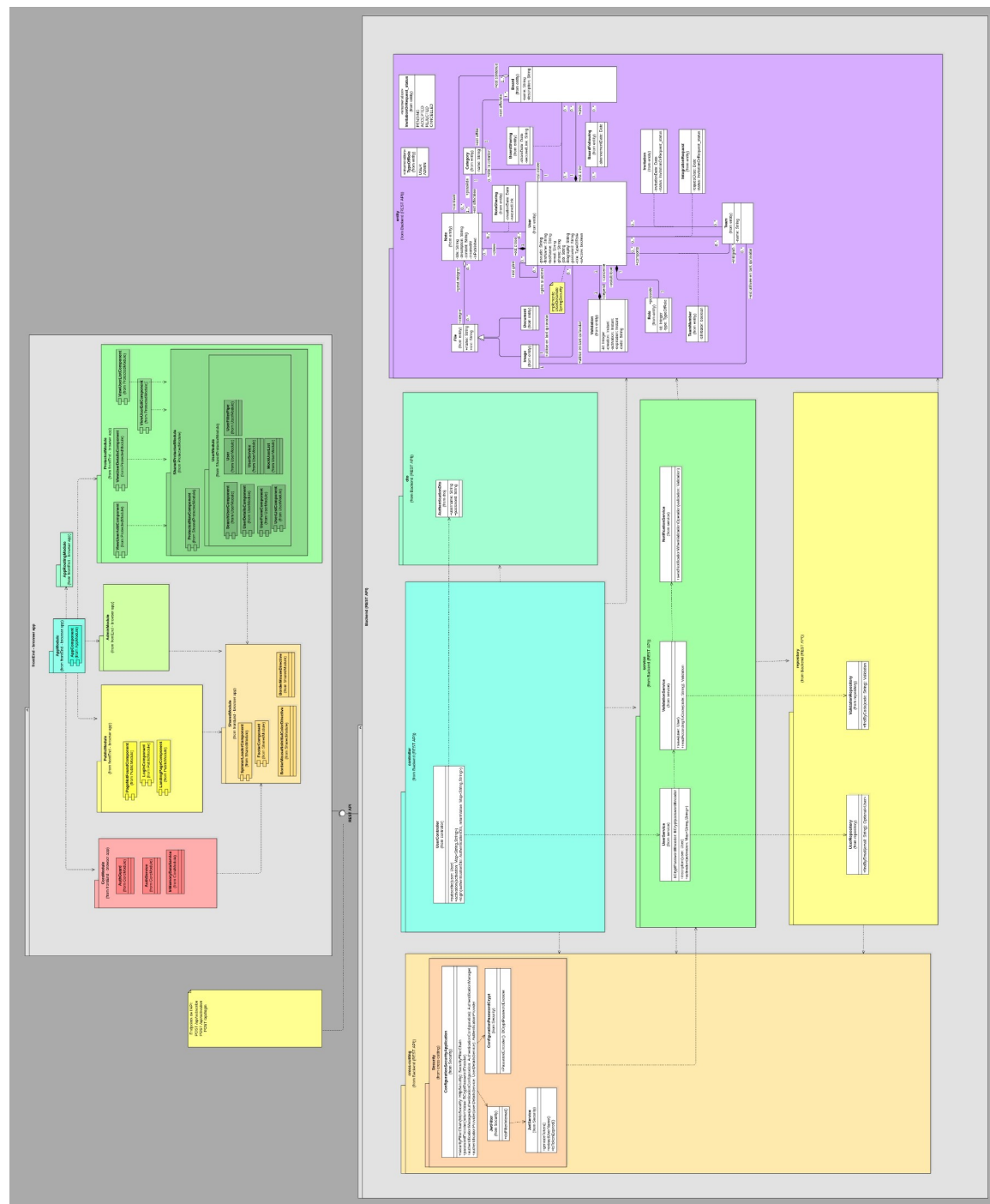


#### **4.3    *Modèle Logique de données (mapping MCD)***

#### **4.4    *Script SQL de création de la base de données***

#### **4.5    *Architecture 3-Tiers de application (couches)***





### Règles de développement côté back

- Respect des conventions de nommage
- Présence de la Javadoc
- Formatage du code
- Les classes entités seront implémentées sous forme de POJO, selon la convention Javabeen.

Exemple :

The diagram illustrates the mapping between the JavaBean rules and the implementation of the `Commune` class. On the left, a green box titled "Modèle JavaBean à que nous nous fixons pour écrire nos classes entités" lists seven rules. On the right, a dark box shows the code for the `Commune` class. Red arrows connect the rules to the corresponding code elements:

- Rule 1: "La classe est simple et ne fait référence à aucun framework particulier" points to the `Commune` class definition.
- Rule 2: "La classe ne doit pas être déclarée final" points to the `public` keyword.
- Rule 3: "La classe possède un Id de type non primitif (Integer par exemple)" points to the `private Integer id;` field.
- Rule 4: "Les propriétés sont privées et exposées par des getters/setters" points to the `private` keyword.
- Rule 5: "forme getX(), setX(), isX()" points to the `Commune()` constructor.
- Rule 6: "Présence d'un constructeur sans arguments" points to the `Commune()` constructor.
- Rule 7: "La classe est Serializable" points to the `implements Serializable` interface.
- Rule 8: "La classe implémente la surcharge des méthodes equals() et hashCode()" points to the `@EqualsAndHashCode` annotation.

The code for the `Commune` class is as follows:

```
@EqualsAndHashCode
@Data
@AllArgsConstructor
public class Commune implements Serializable {

    private Integer id;

    private String nom;

    private Integer codePostal;

    public Commune() {
    }
}
```

- Les controllers ont interdiction d'envoyer les entités sur le réseau : toute donnée envoyée appartenant à une entité doit passer par un DTO si elle sort de l'API. Les DTO ne seront implémentés que par la couche controller.
- Le type record de java sera utilisé pour les dto.

Exemple :

```
public record Person(
    Integer id,
    String lastName,
    String firstName,
    int age
) {
}
```

- Les DTO ne seront utilisés que par la couche controller, à l'intérieur de l'API seront manipulées les entités.
- Les dépendances seront injectées dans les classes et gérées par le conteneur IOC de Spring (utilisation de `@Autowired`)
- Exemple de repository :

Déclare la classe en tant que **Bean Spring** en permettant d'indiquer qu'elle fournit un mécanisme de stockage, d'extraction, de recherche, de mise à jour et de suppression d'objets en base de données.

Interface fournie par **Spring Data Jpa** (héritant de **CrudRepository**). Fournie un ensemble de méthodes taillées sur mesure pour interagir avec une base de données relationnelle. Elle possède deux paramètres génériques à renseigner, le type d'entité associé et le type de son identifiant (future clé primaire)

```
@Repository
public interface AdresseRepository extends JpaRepository<Adresse, Integer> {
}
```

- Exemple de service

L'annotation **@Service** est un alias de **@Component** pour déclarer la classe en tant que **bean Spring** (et donc géré par le conteneur IoC). Elle n'apporte qu'une valeur sémantique. On l'utilise dans la couche service pour indiquer que c'est une classe qui implémente la logique métier.

Gestion de l'injection des dépendances par le conteneur IoC

Indique au conteneur que la méthode s'effectue dans un contexte transactionnel

```
@Service
@AllArgsConstructor
@Data
public class AdresseService {

    @Autowired
    private AdresseRepository adresseRepository;
    @Autowired
    private CommuneService communeService;

    public AdresseService() {}

    @Transactional
    public Adresse findAdresseById(Integer adresseId) {
        Optional<Adresse> adresse = this.adresseRepository
            .findById(adresseId);
        if (adresse.isPresent()) {
            return adresse.get();
        } else {
            throw new Error("Address does not exist");
        }
    }
}
```

- Exemple de controller





## 5 TESTS ET INTÉGRATION

### 5.1 Stratégie de tests unitaires

- De façon générale, toutes les méthodes codées devraient comporter un test unitaire.
- Les repositories n'auront pas à être testés (code Hibernate)
- Tous les services devront comporter un test unitaire
  - Toutes les dépendances utilisées par la classe seront simulées à l'aide de doublure Mockito
  - Exemple de test d'un service :

L'annotation `@Mock` est utilisée pour créer et injecter des instances simulées. Nous ne créons pas d'objets réels, mais nous demandons à Mockito de créer un simulacre de la classe.

Dans Mockito, nous devons créer l'objet d'une classe de test à tester, puis insérer les dépendances simulées pour tester complètement le comportement. Pour ce faire, nous utilisons l'annotation `@InjectMocks`.

Cette dernière permet de marquer un champ sur lequel l'injection doit être effectuée (par injection de constructeur, injection de setter ou injection de propriété).

`@Test` indique à JUnit que la méthode à laquelle elle est attachée peut être exécutée en tant que cas de test.

Définition du comportement de la doublure du repository à adopter pour ce test

Lancement de la méthode du service, et vérification des valeurs retournées par le traitement

On s'assure que la doublure du service a bien été appelée, une unique fois. S'utilise en général en fin de test

```
@RunWith(MockitoJUnitRunner.class)
public class AdresseServiceTest {

    @Mock
    AdresseRepository adresseRepository;

    @InjectMocks
    AdresseService adresseService;

    @Test
    public void adresseService_findAdresseById2_ShouldReturnAdresse() throws Exception {
        Commune communeRef = Commune.builder()
            .nom(nom:"Tulle")
            .codePostal(codePostal:1900)
            .build();

        Adresse adresseRef = Adresse.builder()
            .numero(numero:19)
            .voie(voie:"rue de la Corrèze")
            .commune(communeRef)
            .build();
        adresseRef.setId(id:19);

        Optional<Adresse> adresseOpt = Optional.of(adresseRef);

        // given(this.adresseRepository.findById(19)).willReturn(adresseOpt);
        Mockito.when(this.adresseRepository.findById(adresseRef.getId()))
            .thenReturn(adresseOpt);

        Adresse searchedAddress = adresseService.findAdresseById2(adresseRef.getId());
        assertThat(searchedAddress).isEqualTo(adresseOpt.get());

        Mockito.verify(this.adresseRepository, times(1)).findById(adresseRef.getId());
    }
}
```

Lorsqu'une classe est annotée avec `@RunWith` ou étend une classe annotée avec `@RunWith`, JUnit invoquera la classe qu'elle référence pour exécuter les tests dans cette classe au lieu du runner intégré à JUnit. Mockito JUnit Runner maintient les tests propres et améliore l'expérience de débogage

Création d'une adresse de référence. Cet objet va nous permettre de connaître les valeurs attendues en sortie de test

- Tous les controllers devront comporter un test unitaire
  - Exemple de test d'un controller

Annotation utilisée pour les test Spring MVC qui ne se concentrent uniquement que sur les composants Spring MVC. L'utilisation de cette annotation désactivera l'auto-configuration complète et n'appliquera que la configuration pertinente pour les tests MVC. Par défaut, les tests annotés avec @WebMvcTest configureront également automatiquement Spring Security et MockMvc. Attention : pour les versions de JUnit <5, doit être utilisée avec l'annotation @RunWith(SpringRunner.class)

l'option controllers spécifie les controllers à tester

Active l'auto-configuration de MockMvc. Aucun filtres à appliquer.

On étend les possibilités des JUnit 5 avec l'extension Mockito

```

@WebMvcTest(controllers = AdresseController.class)
@AutoConfigureMockMvc(addFilters = false)
@ExtendWith(MockitoExtension.class)
public class AdresseControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private JWTUtils jwtUtils;
    @MockBean
    private JWTConfig jwtConfig;
    @MockBean
    private RedisUtils redisUtils;
    @MockBean
    private AdresseService adresseService;
    @MockBean
    private CommuneService communeService;

    private Adresse adresseRef;

    @BeforeEach
    public void init() {
        Commune communeRef = new Commune(nom:"Tulle", codePostal:19000);
        adresseRef = new Adresse(numero:19, voie:"rue de la Corrèze", communeRef);
        adresseRef.setId(id:19);
    }

    //Ecriture des tests unitaires
  
```

Injection de MockMvc. Cette classe permet de traiter le code comme s'il traitait une vraie requête HTTP, en dispensant le démarrage du serveur WEB embarqué

Injection de doublures (mocks) grâce à l'annotation @MockBean. Pour un fonctionnement correct, nous devons mocker les trois éléments de sécurité. Nous mockons le service AdresseService, ce qui va permettre de le simuler et de vraiment faire du test unitaire : seule la couche contrôleur doit être testée. Comme adresseService appelle communeService, ce dernier doit aussi être injecté en doublure.

Création d'une adresse de référence, qui sera utilisée par la doublure de AdresseService. Nous l'initialisons dans une méthode annotée @BeforeEach : elle sera ainsi disponible pour les autres cas de test

Pr

@Test indique à JUnit que la méthode à laquelle elle est attachée peut être exécutée en tant que cas de test

Comportement à simuler par la doublure de AdresseService à l'aide des méthode when() et thenReturn() de Mockito

Envoi de la requête GET

Contrôle du status de la réponse http retournée

Contrôle du format de données reçu

Vérification de l'intégrité des données retournées

Envoi du résultat de la requête HTTP sur la sortie standard

On s'assure que la doublure du service a bien été appelé, une unique fois. S'utilise en général en fin de test

```
@Test
public void testGetShouldReturn200OK() throws Exception {
    Mockito.when(this.adresseService.findAdresseById(adresseRef.getId()))
        .thenReturn(adresseRef);

    mockMvc.perform(get("/adresse/{id}", adresseRef.getId()))
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.id").value(adresseRef.getId()))
        .andExpect(jsonPath("$.commune").value(adresseRef.getCommune().getNom()))
        .andExpect(jsonPath("$.codePostal").value(adresseRef.getCommune().getCodePostal()))
        .andExpect(jsonPath("$.numero").value(adresseRef.getNumero()))
        .andExpect(jsonPath("$.voie").value(adresseRef.getVoie()))
        .andDo(print());

    Mockito.verify(this.adresseService,
        times(1)).findAdresseById(adresseRef.getId());
}
```

## 5.2 Stratégie de tests d'intégration

- Tous les Endpoints de l'application devront à minima être testé manuellement via l'interface open-API (Swagger)

The image shows a Swagger UI interface for testing an API endpoint. At the top, it displays the HTTP method 'GET' and the endpoint path '/adresse/{adresseId}'. Below this, there is a 'Parameters' section. A table with two columns, 'Name' and 'Description', lists the parameters. One parameter is shown: 'adresseId', which is marked as 'required' with a red asterisk. Its description is 'integer(\$int32)' and '(path)'. To the right of the parameter name is a text input field containing the value '8'. At the bottom right of the interface is a blue button labeled 'Execute'.

Name	Description
adresseId <small>* required</small>	integer(\$int32) (path)

- Voir pour l'automatisation du test des Endpoint