

# Présentation de projet Concepteur Développeur d'Applications

Du 16 mars au 2 décembre 2022

Guillaume Andreux

CDA 21260-21324

Centre de formation Afpa, Paris Politzer

# Sommaire

Introduction ..... p. 4

## I Environnement de travail

1. Langages et base de données ..... p. 7
2. Outils ..... p. 7

## II Analyse des besoins

1. Analyse globale de l'application ..... p. 9
2. Diagrammes des cas d'utilisation ..... p. 10
3. Raffinements des cas d'utilisation ..... p. 13
4. Diagrammes de séquence ..... p. 20

## III Conception

1. Diagramme de classe ..... p. 24
2. Modèle Conceptuel de Données ..... p. 27
3. Modèle Physique de Données ..... p. 28
4. Structure de la base de données ..... p. 29

5. Wireframes ..... p. 31

6. Maquette ..... p. 32

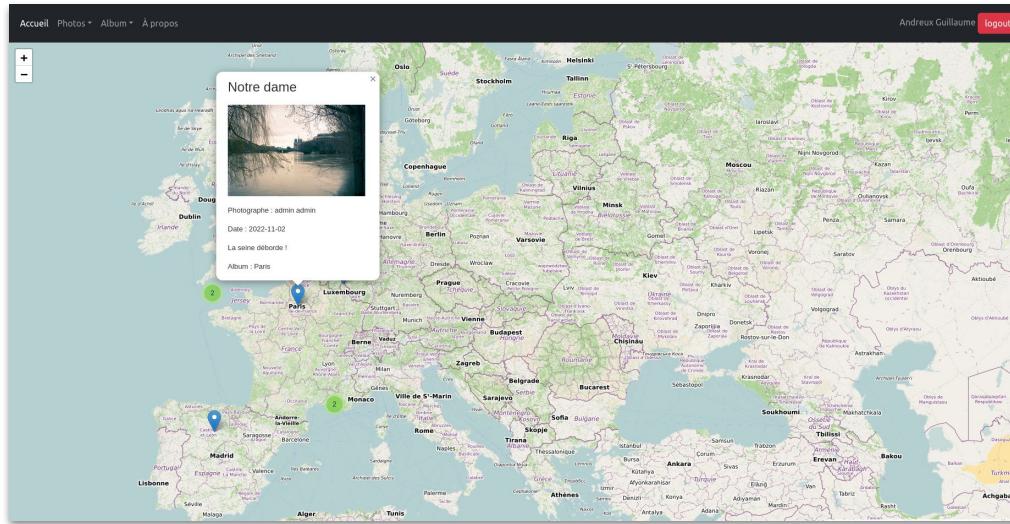
## IV Développement

1. Technologies utilisées ..... p. 34
2. Architecture Spring Boot ..... p. 37
3. Configuration du projet ..... p. 39
4. Accès aux données ..... p. 42
5. Spring Data JPA ..... p. 44
6. Couche métier ..... p. 47
7. Couche de présentation
  - 7.1. Controller ..... p. 48
  - 7.2. Templates Thymeleaf ..... p. 51
8. Focus
  - 8.1. Import de fichiers ..... p. 54
  - 8.2. Affichage des fichiers images ..... p. 56
  - 8.3. Bibliothèque Leaflet ..... p. 57
9. Spring Security ..... p. 60
10. Tests unitaires ..... p. 66

V Affichage dans le navigateur ..... p. 70

Conclusion ..... p. 78

# APPLICATION WEB GÉOLOCPHOTOS



# INTRODUCTION

# Présentation du projet

J'ai réalisé un stage en entreprise qui ne m'a pas permis de mettre pleinement en pratique les notions apprises pendant la formation.

Je vous présente donc un projet personnel développé en parallèle de ce stage. Il s'agit d'une application web permettant de publier ses propres photos.

Cette application web, développée en Programmation Orientée Objet, reprend les notions apprises pendant la formation :

- conception d'une application ;
- conception d'une base de données ;
- développement de composants métiers ;
- développement d'une interface Frontend.

Je tiens à remercier Moussa Camara, notre formateur pendant cette formation qui a su pendant ces quelques mois nous apprendre les métiers de développeur web.

# ENVIRONNEMENT DE TRAVAIL

## I.1 Langages et base de données

- **JavaSE** dans la version OpenJDK 17.0.4
- **SQL**
- **Base de données** : MySQL version 8.0.31
- **Javascript**
- **HTML 5**
- **CSS 3**

## I.2 Outils

- Conception UML : **StarUML**
- Conception Merise : **Analyse SI**
- wireframes : **Pencil**
- maquette : **Figma**
- éditeur de code statique : **Visual Studio Code**
- IDE : **Eclipse**
- système de gestion de version : **Git**
  - en ligne de commande : **Git bash**
  - interface git en mode console : **Tig**

# ANALYSE DES BESOINS

## II.1 Analyse globale de l'application

L'objectif est de concevoir et développer une application permettant d'afficher des photos avec des métadonnées y afférant. Deux interfaces seront possibles :

- une interface classique avec des images miniatures qu'il est possible d agrandir;
- une interface « géographique », où les photos seront localisées sur une carte géographique par un marqueur cliquable permettant de visualiser la photo et ses métadonnées.

Il sera possible pour un utilisateur inscrit d'y téléverser ses propres photos et d'ajouter les métadonnées.

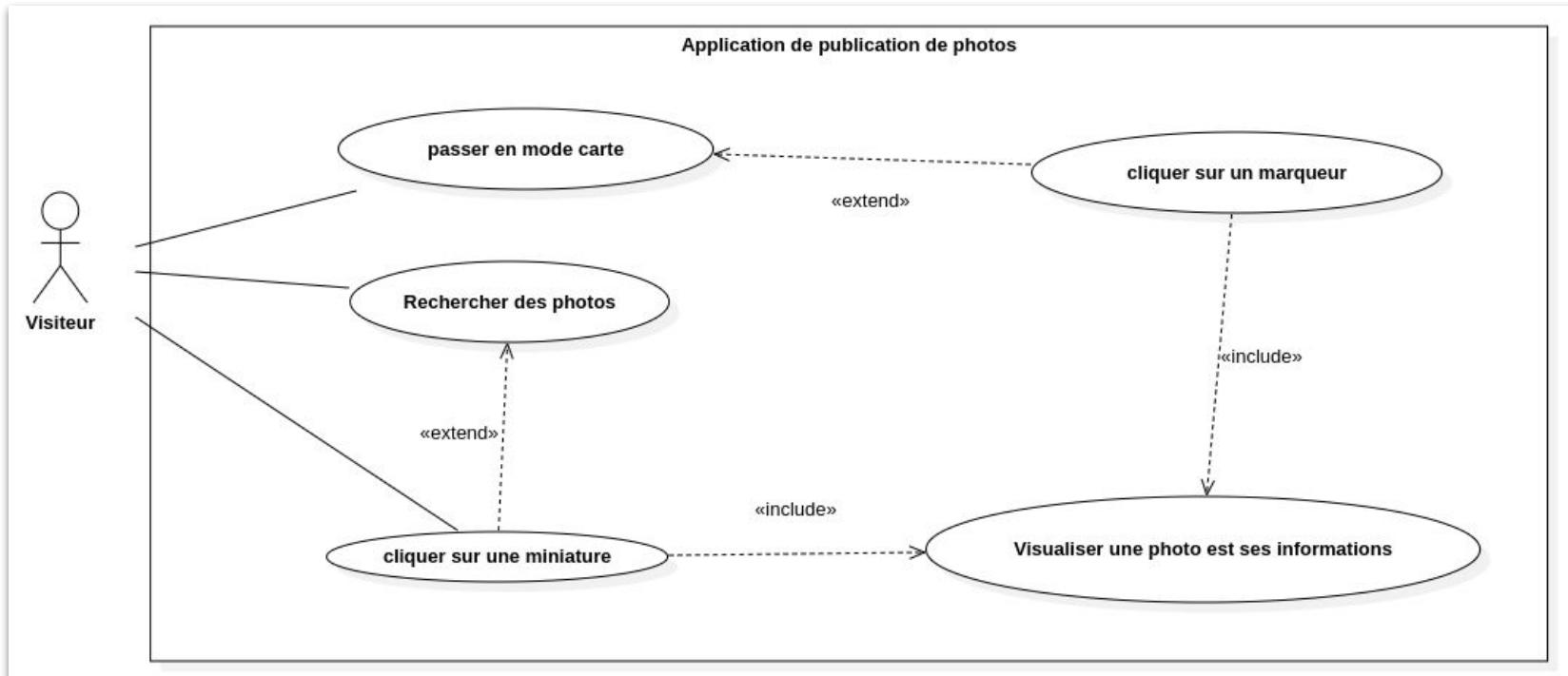
### Description des acteurs

**Le visiteur** navigue sur le site en se déplaçant sur la carte ; il clique sur des marqueurs pour consulter les photos ; il sélectionne une catégorie pour filtrer les photos.

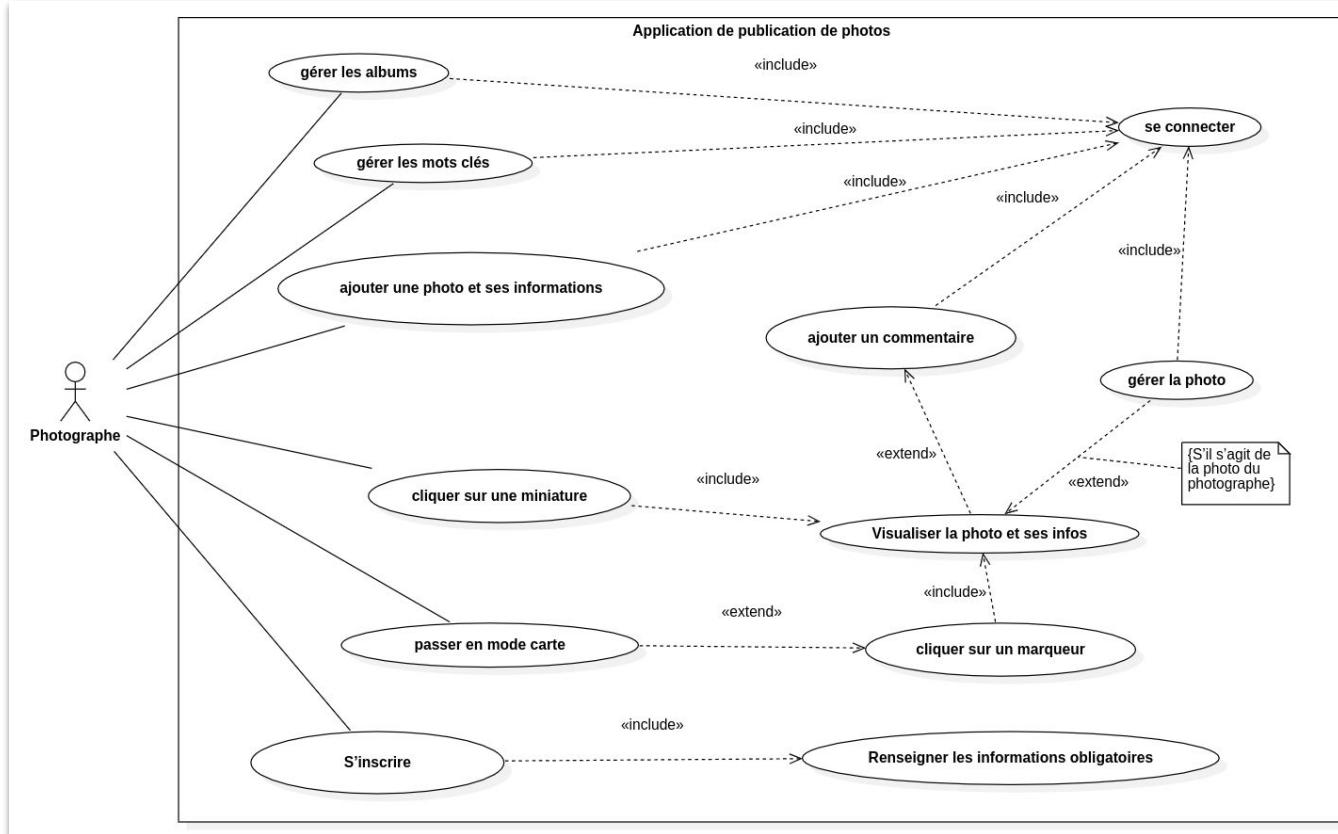
**Le photographe** peut ajouter, modifier, supprimer ses photos et ses albums. Il peut aussi ajouter des commentaires.

**L'administrateur** peut ajouter, modifier, supprimer des catégories ; il peut consulter la liste des tous les photographes, de tous les albums, de toutes les catégories, de tous les mots-clés et de toutes les photos avec leurs commentaires.

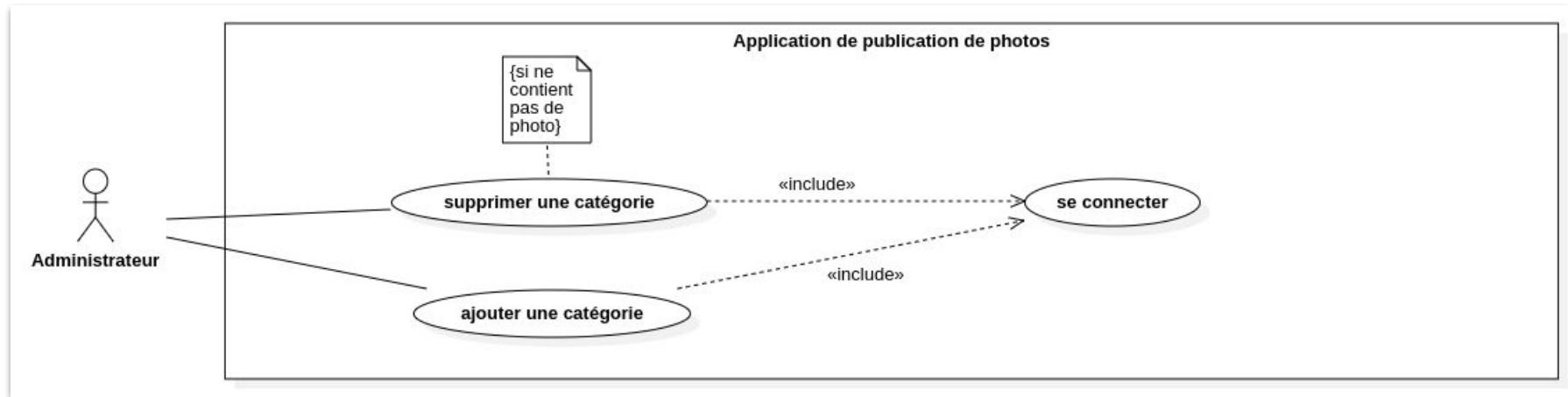
## II.2 Diagrammes des cas d'utilisation : visiteur



## II.2 Diagrammes des cas d'utilisation : photographe



## II.2 Diagrammes des cas d'utilisation : administrateur



## II.3 Raffinement des cas d'utilisation : inscription

<b>Nom du cas d'utilisation</b>	Inscription	<b>Scénario principal</b>	1-Le système affiche le formulaire d'inscription 2-le système demande à l'utilisateur (photographe) de saisir son adresse email et son mot de passe.
<b>Acteurs</b>	Photographe		
<b>Pré condition</b>	Le système fonctionne et l'utilisateur veut bénéficier des services offerts par l'application	<b>Exception</b>	
<b>Post condition</b>	L'utilisateur aura un mot de passe et un login et pourra accéder aux services de l'application.		L'adresse email existe déjà dans le système. Dans ce cas un message d'erreur sera émis invitant le client à saisir de nouveau une adresse email.

## II.3 Raffinement des cas d'utilisation : authentification

<b>Nom du cas d'utilisation</b>	Authentification	<b>Scénario principal</b>	1-Le système affiche le formulaire d'authentification 2-L'utilisateur saisit son adresse email et mot de passe 3-l'utilisateur bénéficiera des fonctionnalités de l'application selon son privilège (administrateur/photographe) 4-Le système affiche un message d'erreur si l'adresse email ou le mot de passe sont incorrects.
<b>Acteurs</b>	Administrateur, photographe		
<b>Pré condition</b>	Chaque personne qui désire bénéficier des services de l'application doit s'authentifier. Cette personne doit saisir son adresse email et son mot de passe.	<b>Exception</b>	
<b>Post condition</b>	Chaque utilisateur accédera aux fonctionnalités de l'application selon ses priviléges.		En cas d'inexistence du mot de passe et de l'adresse email introduits un message d'erreur est affiché

## II.3 Raffinement des cas d'utilisation : gestion des catégories

<b>Nom du cas d'utilisation</b>	Gestion des catégories	<b>Scénario principal</b>	1-Le système présente une interface qui se compose de deux boutons ajouter et gérer une catégorie 2-l'administrateur pourra ajouter une catégorie 3-Dans le cas d'ajout, l'administrateur doit insérer le nom de la catégorie 4-l'administrateur peut modifier ou supprimer une catégorie.
<b>Acteurs</b>	Administrateur		
<b>Pré condition</b>	Le système fonctionne et l'administrateur est authentifié.	<b>Exception</b>	Le système affiche un message d'erreur dans les cas où : - pour la suppression, si la catégorie contient des photos - pour l'ajout, si la catégorie existe déjà dans la base des données.
<b>Post condition</b>	L'administrateur pourra ajouter, supprimer et modifier les catégories.		

## II.3 Raffinement des cas d'utilisation : gestion des albums

<b>Nom du cas d'utilisation</b>	Gestion des albums	<b>Scénario principal</b>	1-Le système présente une interface qui se compose de deux boutons ajouter et gérer un album 2-le photographe pourra ajouter un album 3-Dans le cas d'ajout, le photographe doit insérer le nom de l'album 4-le photographe peut modifier ou supprimer un album.
<b>Acteurs</b>	Photographe		
<b>Pré condition</b>	Le système fonctionne et le photographe est authentifié	<b>Exception</b>	Le système affiche un message d'erreur dans les cas où : - pour la suppression, si l'album contient des photos - pour l'ajout, si l'album existe déjà dans la base des données.
<b>Post condition</b>	Le photographe pourra ajouter, supprimer et modifier les albums.		

## II.3 Raffinement des cas d'utilisation : gestion des photos

<b>Nom du cas d'utilisation</b>	Gestion des photos	<b>Scénario principal</b>	1-Le système présente une interface qui se compose de deux boutons ajouter et gérer une photo 2-le photographe pourra ajouter une photo 3-Dans le cas d'ajout, le photographe doit insérer un titre, une description, une localisation, un fichier image, choisir un album, une ou plusieurs catégories, le statut public ou privé, peut ajouter des mots clés 4-le photographe peut modifier les informations ou supprimer une photo.
<b>Acteurs</b>	Photographe		
<b>Pré condition</b>	Le système fonctionne et le photographe est authentifié		
<b>Post condition</b>	Le photographe pourra ajouter, supprimer et modifier les photos.	<b>Exception</b>	Le système affichera un message d'erreur si un ou plusieurs champs obligatoires ne sont pas remplis.

## II.3 Raffinement des cas d'utilisation : visualisation des photos

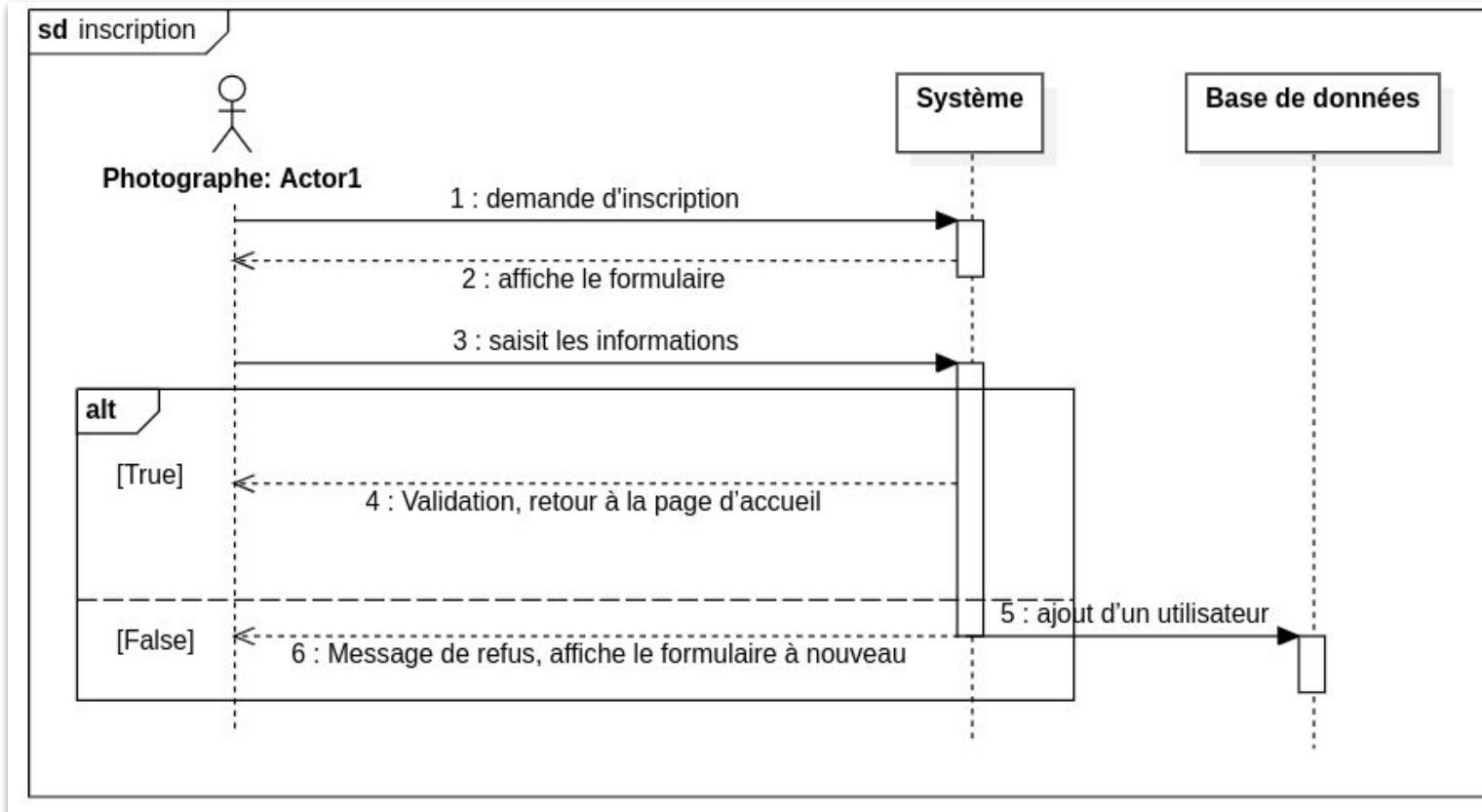
<b>Nom du cas d'utilisation</b>	Gestion des photos	<b>Scénario principal</b>	1-Le système présente une interface qui se compose de la photo, ses informations. 2-le photographe pourra ajouter un commentaire ou un favori 3-Dans le cas d'ajout d'un commentaire, le photographe doit insérer son commentaire dans un champ prévu à cet effet.
<b>Acteurs</b>	Photographe		
<b>Pré condition</b>	Le système fonctionne et le photographe est authentifié		
<b>Post condition</b>	Le photographe pourra ajouter des favoris et des commentaires.	<b>Exception</b>	

## II.3 Raffinement des cas d'utilisation : gestion des mots-clés

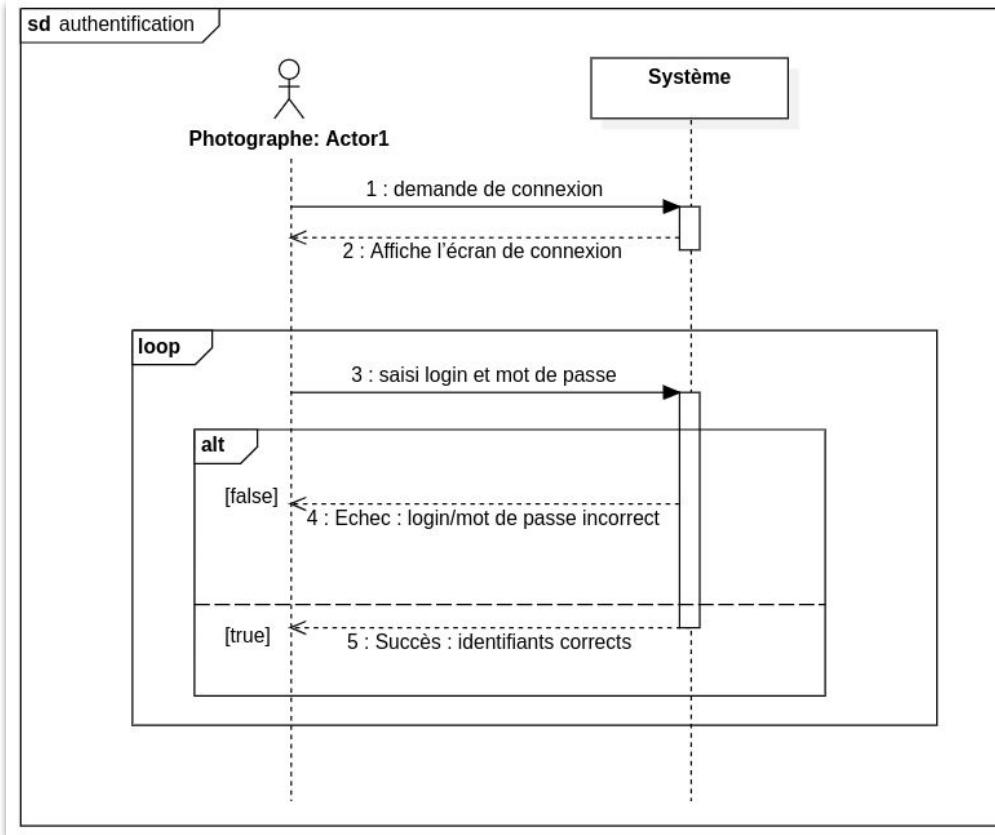
<b>Nom du cas d'utilisation</b>	Gestion des mots-clés	<b>Scénario principal</b>	1-Le système présente une interface qui se compose d'un champ de formulaire pour ajouter des mots-clés. 2-le photographe pourra ajouter des mots-clés 3-l'administrateur peut modifier ou supprimer un mot-clé
<b>Acteurs</b>	Administrateur, Photographe		
<b>Pré condition</b>	Le système fonctionne et le photographe ou l'administrateur est authentifié		
<b>Post condition</b>	Le photographe pourra ajouter des mots-clés. L'administrateur pourra modifier, supprimer des mots-clés.	<b>Exception</b>	

# CONCEPTION

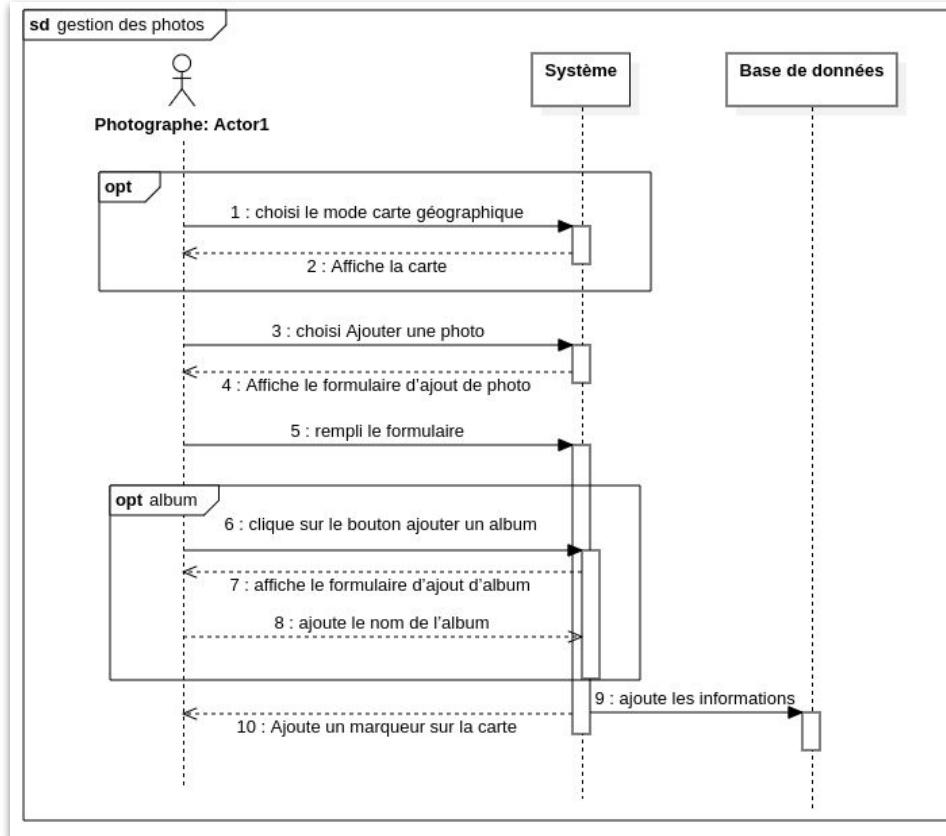
### III.1 Diagramme de séquence : inscription



### III.1 Diagramme de séquence : authentification



### III.1 Diagramme de séquence : ajout d'une photo



## III.2 Diagramme de classes

### Les classes

De l'analyse des besoins, nous pouvons en déduire les classes qui le constituent.

**La classe User** qui aura pour attributs un identifiant, le nom, le prénom, l'adresse email, le mot de passe, les latitude et longitude par défaut et le rôle (administrateur ou photographe).

**La classe Photo** qui aura pour attributs un identifiant, le titre, la description, la date, la latitude, la longitude, le nom du fichier image et le fait d'être publique ou non.

**Les classes Category, Keyword, Album** qui auront chacune pour attributs un identifiant et un nom.

La classe **Comment** qui aura pour attribut un commentaire.

## III.2 Diagramme de classes

### Les cardinalités

**User - Photo** : un utilisateur pourra avoir zéro ou plusieurs photos, une photo aura un et un seul utilisateur.

**User - Album** : un utilisateur pourra avoir zéro ou plusieurs albums, un album aura un et un seul utilisateur.

**User - Category** : un utilisateur (administrateur) pourra avoir zéro ou plusieurs catégories, une catégorie aura un et un seul utilisateur.

**User - Keyword** : un utilisateur pourra ajouter zéro ou plusieurs mots-clés, un mot-clé aura un et un seul utilisateur.

**User - Comment** : un utilisateur pourra laisser zéro ou plusieurs commentaires, un commentaire sera laissé par un et un seul utilisateur.

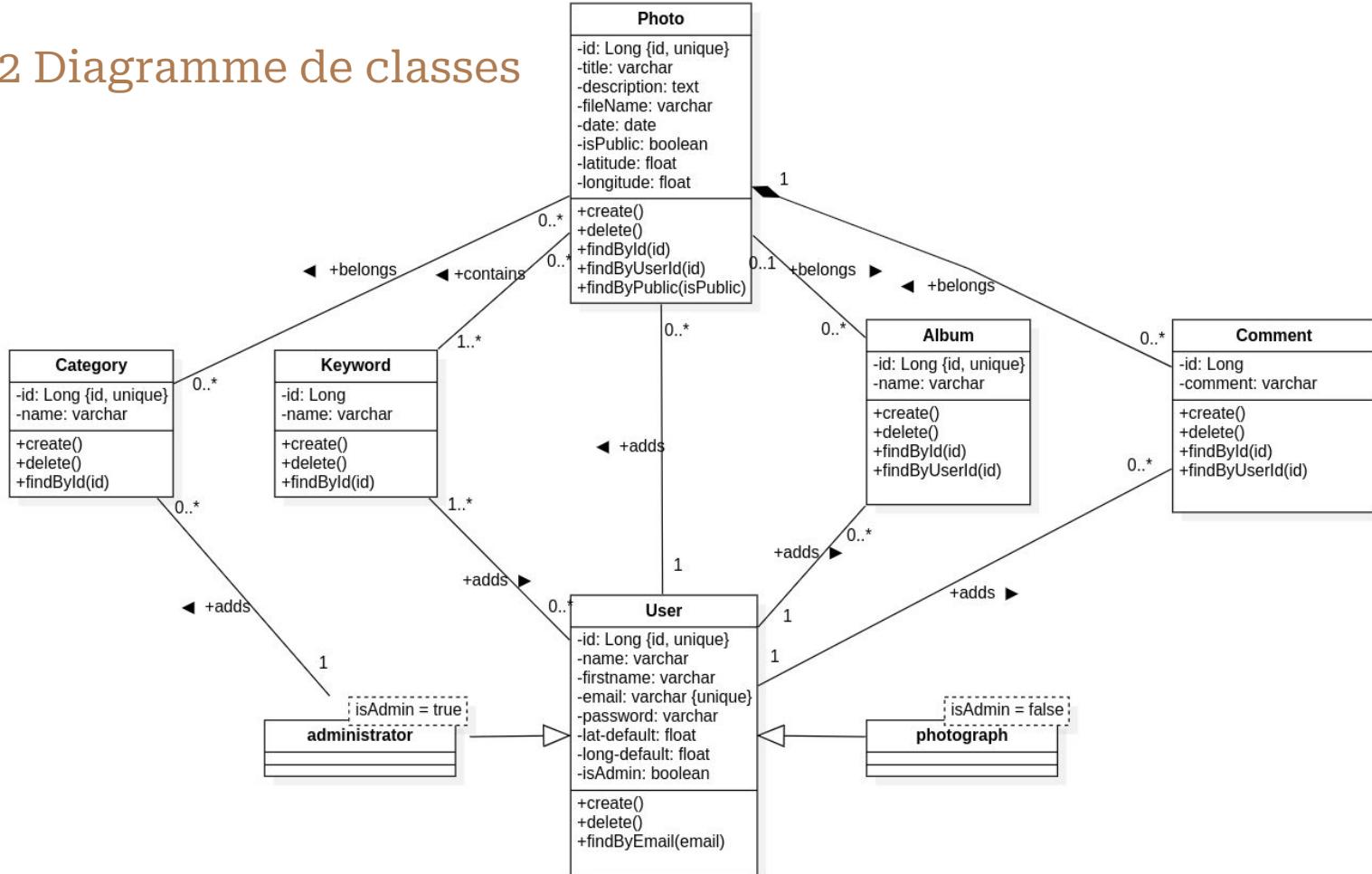
**Album - Photo** : un album pourra avoir zéro ou plusieurs photos, une photo aura zéro ou un album.

**Category - Photo** : une catégorie pourra avoir zéro ou plusieurs photos, une photo aura zéro ou plusieurs catégories.

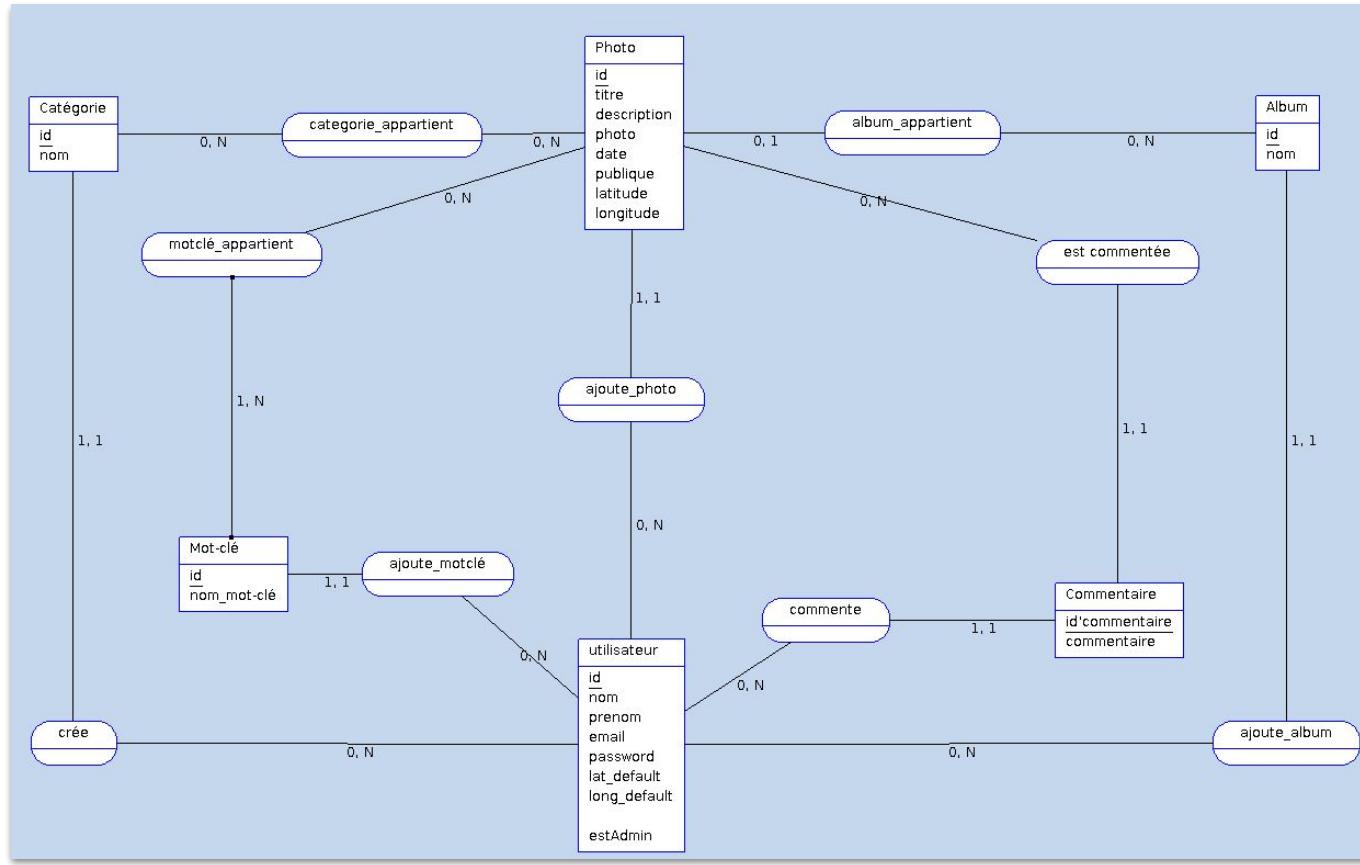
**Keyword - Photo** : un mot-clé pourra avoir une ou plusieurs photos, une photo aura zéro ou plusieurs mots-clés.

**Comment - Photo** : un commentaire sera pour une et une seule photo, une photo aura zéro ou plusieurs commentaires.

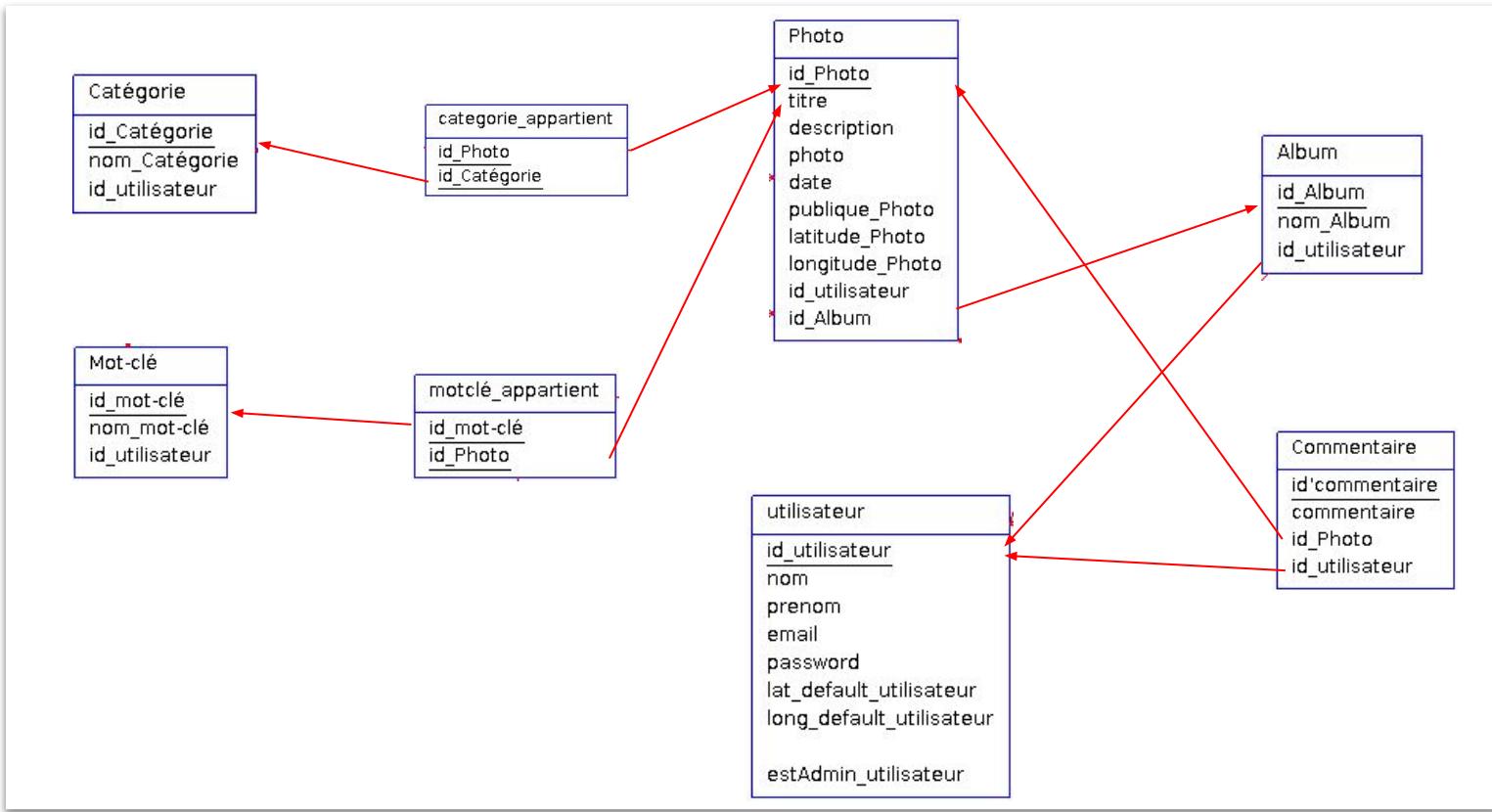
## III.2 Diagramme de classes



### III.3 Modèle Conceptuel de données



### III.4 Modèle Physique de données



### III.5 Structure de la base de données

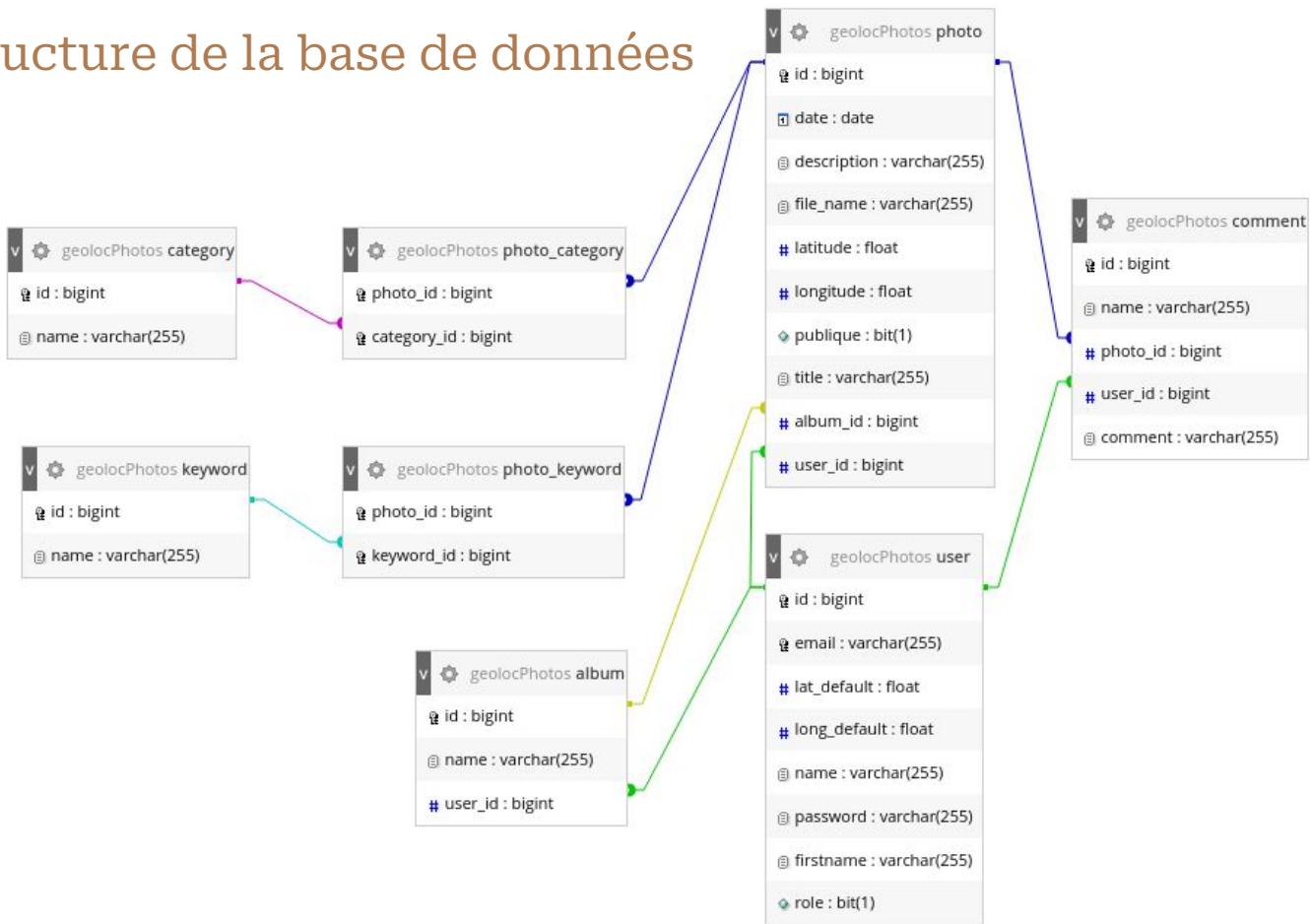
Avec UML, du diagramme de classe et des cardinalités entre les classes, nous pouvons en déduire les tables que contiendra la base de données.

De chaque classe découlera une table. La base de données contiendra donc **les tables photo, category, album et keyword**.

Des relations plusieurs à plusieurs entre d'une part les tables category et photo et d'autre part les tables keyword et photo, découlent pour chacune de ces relations **les tables intermédiaires photo\_keyword et photo\_category**.

Avec Merise, le modèle physique de données nous donne directement le schéma de la base de données.

### III.5 Structure de la base de données



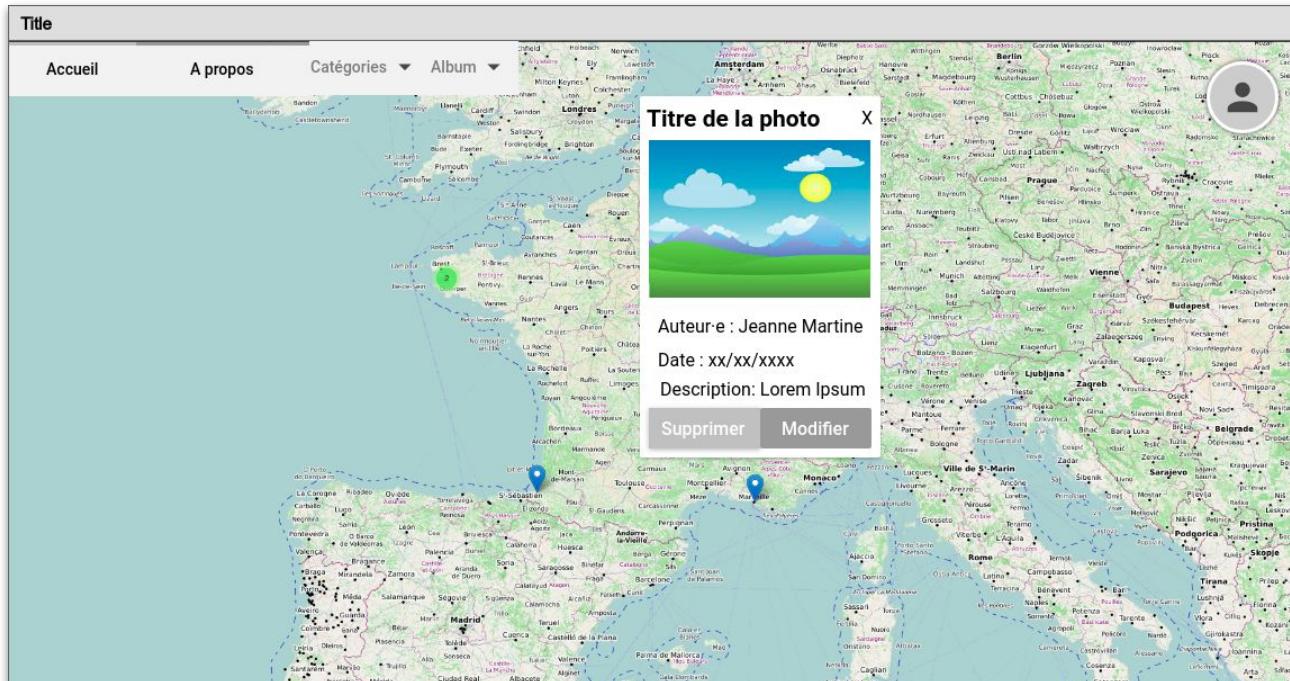
## III.6 Wireframe

### Wireframe de la page d'affichage d'une photo

Les wireframes permettent de créer une maquette simplifiée où sont placés les différents éléments de la page indépendamment de la question graphique.

Comme exemple, voici celui de la page d'affichage d'une photo.

Les wireframes ont été réalisés avec le logiciel Pencil



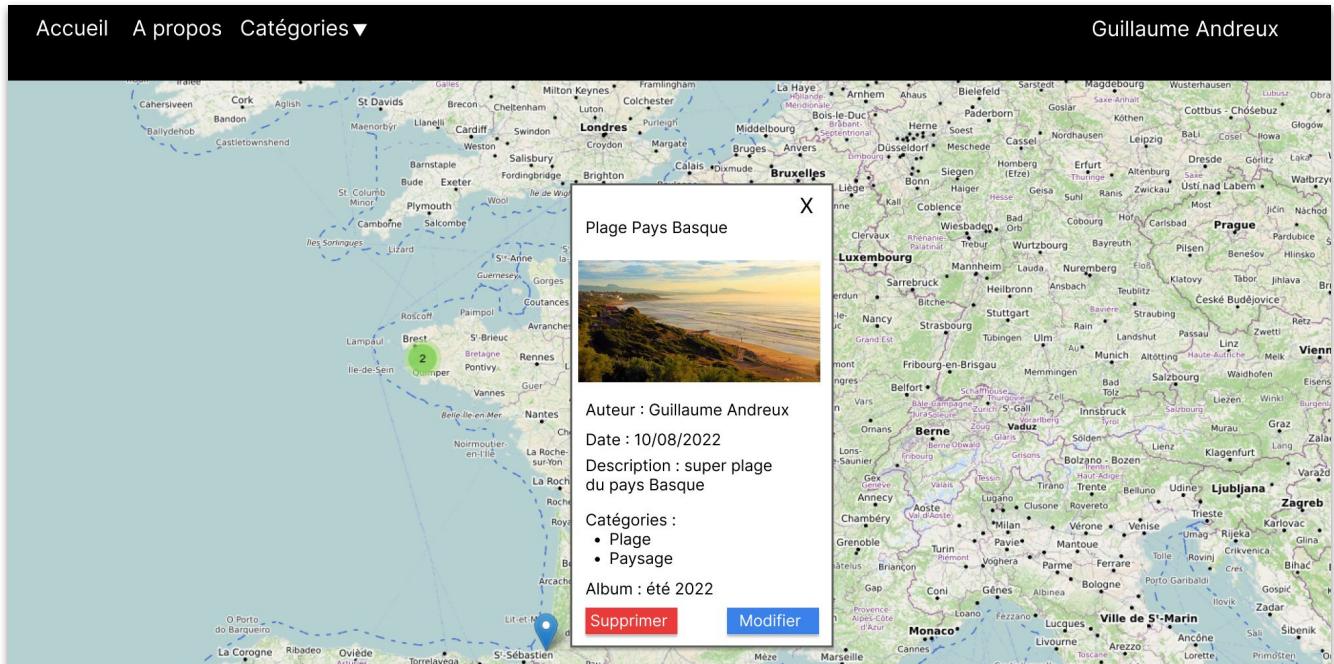
## III.7 Maquette

### Maquette de la page d'affichage d'une photo

L'étape d'après consiste à reprendre les wireframes pour y ajouter la couche graphique.

Comme exemple, voici celle de la page d'affichage d'une photo.

Les maquettes ont été réalisées avec le logiciel Figma.



# DÉVELOPPEMENT

## IV.1 Technologies utilisées

### Back End

L'application est développée en JavaEE.

- **Spring Boot** 2.7.5 : Extension du framework d'application Spring. Spring Boot permet d'installer, de configurer et d'exécuter des applications plus facilement et plus rapidement avec moins de configuration.
  - **Java Persistence API (JPA)** est une norme Java qui permet de lier des objets Java à des enregistrements dans une base de données relationnelle. JPA permet de récupérer, stocker, mettre à jour et supprimer des données dans une base de données relationnelle à l'aide d'objets Java.

Les annotations JPA décrivent comment une classe Java donnée et ses variables correspondent à une table donnée et ses colonnes dans une base de données.

- **Spring Security** est une librairie qui permet de préconfigurer et de personnaliser des fonctions de sécurité au sein d'une application Java. Il est possible d'utiliser Spring Security pour permettre une connexion de manière sécurisée, ou s'assurer que les bons utilisateurs disposent du niveau d'accès approprié à l'application.

## IV.1 Technologies utilisées

- **Spring Boot** 2.7.5 suite
  - **Hibernate** est un framework ORM (Object / Relational Mapping) qui concerne la persistance des données. Il s'agit d'une solution de mappage objet-relationnel qui mappe les classes Java aux tables dans les bases de données relationnelles et des types de données Java à SQL. Il permet d'obtenir de la persistance.
  - **Thymeleaf** est un moteur de rendu de document écrit en Java. Principalement conçu pour produire des vues Web, en HTML, JavaScript et CSS .
- **Maven** est un outil de gestion et d'automatisation de production des projets logiciels Java. Chaque projet est configuré par un Project Object Model (POM) qui contient les informations nécessaires à Maven pour traiter le projet (nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, noms des contributeurs, etc.). Ce POM se matérialise par un fichier pom.xml à la racine du projet.

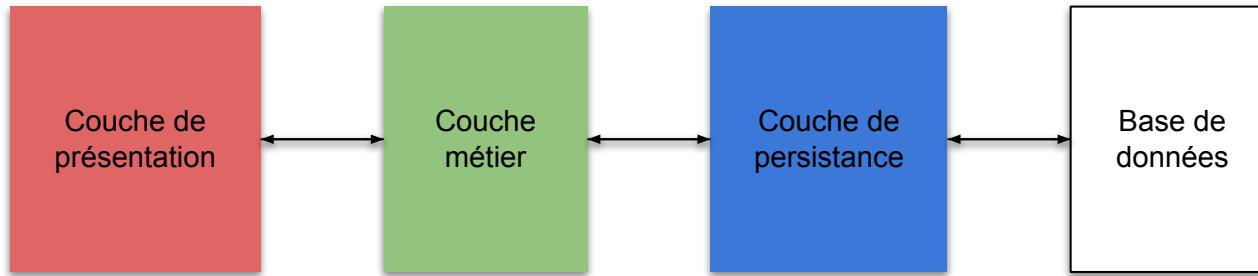
## IV.1 Technologies utilisées

### Front End

Dans l'état actuel de développement de l'application, seule l'interface « géographique » existe. Par conséquent, cette application, basée sur un affichage cartographique, est essentiellement écrite en Javascript.

- **HTML5**
- **CSS3**
- **Bibliothèques Javascript :**
  - **Jquery 3.6.1.**
  - **Leaflet 1.8.0** : une bibliothèque JavaScript libre de cartographie en ligne. Elle est utilisée avec OpenstreetMap.
- **Framework CSS :**
  - **Bootstrap 5.2.1**

## IV.2 Architecture Spring Boot



Spring Boot suit une **architecture n-tiers** en couches dans laquelle chaque couche communique avec la couche directement en dessous ou au-dessus (structure hiérarchique).

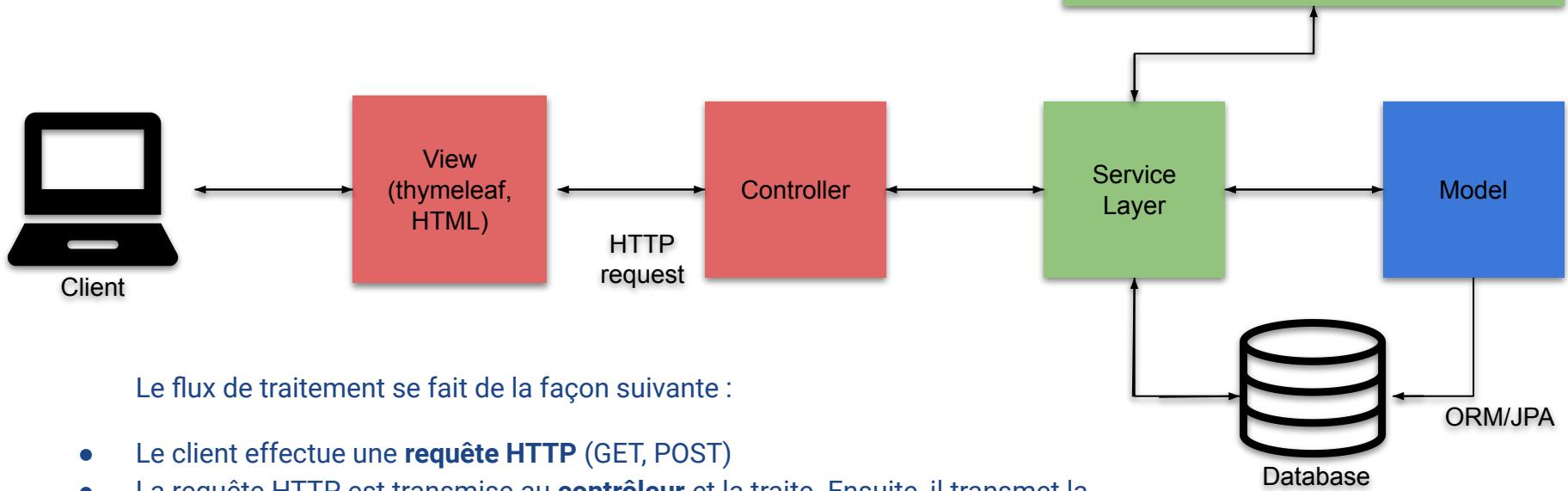
**La couche de présentation** est la couche supérieure de l'architecture. Elle se compose des **contrôleurs**, qui gèrent les **requêtes HTTP** et effectue l'authentification et des **vues**, qui sont envoyées au client.

**La couche métier** contient toute la **logique métier**. Elle se compose de **classes de services**.

**La couche de persistance** contient toute la **logique de stockage** de la base de données. Elle est responsable de la conversion des objets métier en ligne de base de données et vice-versa.

**La couche de base de données** contient la base de données MySQL de l'application.

## IV.2 Architecture Spring Boot



- Le client effectue une **requête HTTP** (GET, POST)
- La requête HTTP est transmise au **contrôleur** et la traite. Ensuite, il transmet la demande à la couche métier si nécessaire.
- Dans la **couche de service**, toute la logique métier s'exécute. Elle exécute la logique sur les données qui sont mappées à JPA avec des **classes de modèle**.
- **La vue Thymeleaf/HTML** est renvoyée en tant que réponse du contrôleur si aucune erreur ne s'est produite.

## IV.3 Configuration du projet

### Initialisation du projet

Le site [Spring Initializr](#) permet d'initialiser le projet en ajoutant notamment les dépendances nécessaires.

Il y a donc les dépendances suivantes sur ce projet qui seront ajoutées au fichier pom.xml :

- Spring web
- Spring Boot DevTools
- Thymeleaf
- Spring Security
- Spring Data JPA
- Validation
- MySQL Driver

## IV.3 Configuration du projet

### Fichier pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.3</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>geolocPhotos</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>geolocPhotos</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
  </dependencies>
```

```
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
      <groupId>org.thymeleaf.extras</groupId>
      <artifactId>thymeleaf-extras-springsecurity5</artifactId>
    </dependency>
  </dependencies>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

## IV.3 Configuration du projet

Le fichier **application.properties** permet de configurer la base de données avec notamment son url, le username et le mot de passe.

Il est possible de spécifier comment Hibernate gère la base de données. Dans ce projet, elle est mise à jour automatiquement.

Il est possible de spécifier un port et un context-path pour l'affichage dans le navigateur.

Il est aussi possible de configurer la dépendance spring-security pour l'accès à l'application.

Enfin pour le téléchargement des fichiers, il est spécifié la taille maximum de chaque fichier.

Fichier *application.properties*

```
## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.url = jdbc:mysql://localhost:3306/geolocPhotos?createDatabaseIfNotExist=true
spring.datasource.username = guillaume
spring.datasource.password = *****

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format-sql=true

## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update

spring.thymeleaf.mode: HTML

server.port=8081
server.servlet.context-path=/geolocPhotos

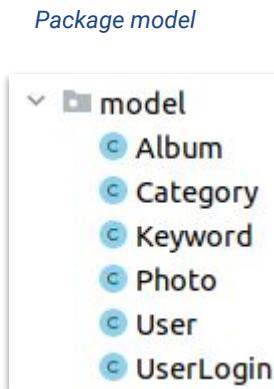
spring.security.user.name=admin
spring.security.user.password=test123

spring.servlet.multipart.max-file-size=128000KB
```

## IV.4 Accès aux données

Dans le **package Model** se trouvent les classes permettant la persistance des données.

À partir du diagramme de classes, on peut construire les classes Entités. Tous les attributs des classes du diagramme seront présents. Les cardinalités permettent de construire les jointures.



Hibernate permet avec le jeu d'annotations de créer les tables, les clés primaires, les clés secondaires et les tables d'associations :

- **@Entity** : crée une table du nom de la classe ;
- **@Table** : définit le nom de la table ;
- **@Id** : déclare l'attribut comme clé primaire unique ;
- **@OneToOne**, **@ManyToOne**, **ManyToMany** : créent des jointures entre les tables ;
- **@JoinColumn(name="nomTable\_id")** : définit le nom de la colonne clé étrangère de la table associée ;
- **@JoinTable(name="nomTable\_nomTableAssociée")** : définit le nom de la table d'association entre deux entités ayant une relation plusieurs à plusieurs.

## IV.4 Accès aux données

### Exemple de la classe Photo du package Model

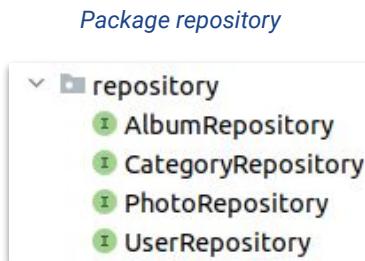
Classe Photo (extrait)

```
23 @Entity
24 public class Photo {
25
26     @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
27     private Long id;
28
29     private String title;
30
31     private String description;
32
33     private String fileName;
34
35     private Date date;
36
37     private boolean publique;
38
39     private float latitude;
40
41     private float longitude;
```

```
43     @ManyToOne
44     @JoinColumn(name="user_id")
45     private User user;
46
47     @ManyToOne
48     @JoinColumn(name="album_id")
49     private Album album;
50
51
52     @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)
53     @JoinTable(
54         name = "photo_category",
55         joinColumns = @JoinColumn(name = "photo_id", referencedColumnName = "id"),
56         inverseJoinColumns = @JoinColumn(name = "category_id", referencedColumnName = "id")
57     )
58     private Set<Category> category = new HashSet<~>();
59
60
61     @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)
62     @JoinTable(
63         name = "photo_keyword",
64         joinColumns = @JoinColumn(name = "photo_id", referencedColumnName = "id"),
65         inverseJoinColumns = @JoinColumn(name = "keyword_id", referencedColumnName = "id")
66     )
67     private Set<Keyword> keyword = new HashSet<~>();
```

## IV.5 Spring Data JPA

**Spring Data** est un projet *Spring* qui a pour objectif de simplifier l'interaction avec différents systèmes de stockage de données. *Spring Data* fournit des interfaces par défaut et définit une convention de nommage des méthodes d'accès pour exprimer la requête à réaliser.



**Spring Data JPA** est le module qui permet d'interagir avec une base de données relationnelles en représentant les objets du domaine métier sous la forme d'entités JPA.

Pour définir un *repository*, il suffit de créer une interface qui hérite d'une *JpaRepository<T, ID>*.

Il faut tout d'abord déclarer la dépendance JPA dans le fichier pom.xml :

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

## IV.5 Spring Data JPA

L'interface `JpaRepository<T, ID>` déclare essentiellement les méthodes CRUD. Elles ne permettent pas d'implémenter toutes les fonctionnalités attendues d'une application.

Pour l'interface Photo, il a fallu ajouter les méthodes équivalentes aux requêtes SQL suivantes :

- **findByPublique** : “`SELECT * FROM Photo WHERE publique IS TRUE`”
- **findById** : “`SELECT * FROM photo WHERE user_id = :user_id`”
- **findByAlbumId** : “`SELECT * FROM photo WHERE album_id = :album_id`”
- **findByCategoryId** : “`SELECT * FROM photo INNER JOIN photo_category ON photo.id = photo_category.photo_id WHERE category_id = :category_id`”

*Interface repository/PhotoRepository*

```
9  public interface PhotoRepository extends JpaRepository<Photo, Long>{  
10  
11      List<Photo> findByPublique(boolean publique);  
12  
13      List<Photo> findById(Long id);  
14  
15      List<Photo> findByAlbumId(Long id);  
16  
17      List<Photo> findByCategoryId(Long id);  
18  }  
19
```

## IV.5 Spring Data JPA

Hibernate permet de créer des requêtes préparées (prepared statements).

`findById` : “*SELECT \* FROM photo WHERE user\_id = :user\_id*” s’écrit de la façon suivante :

```
PreparedStatement sql = connect.prepareStatement("SELECT * FROM photo WHERE user_id = ?");

sql.setInt(1, user_id);

ResultSet rs = sql.executeQuery();

while (rs.next()) {

    Photo photo = new Photo(res.getLong("id"), res.getString("title"), res.getString("description"),
    res.getString("filename"), res.getDate("date"), res.getBoolean("publique"), res.getFloat("latitude"),
    res.getFloat("longitude"));

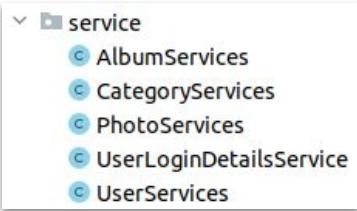
}
```

Une requête préparée permet de protéger l’application **des injections SQL**.

## IV.6 Couche métier

**La couche métier** se situe dans le package Service qui contient des classes qui appellent les interfaces du Repository.

Toutes les méthodes de la classe PhotoServices sont appelées par les contrôleurs et la méthode équivalente de l'interface PhotoRepository qui a un accès direct aux données via les requêtes SQL.



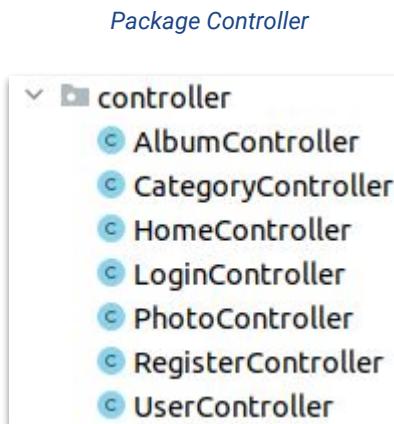
Package service

Classe service/PhotoServices

```
12  @Service
13  public class PhotoServices {
14
15      @Autowired
16      private PhotoRepository photoRepository;
17
18      public Photo createPhoto(Photo photo) { return photoRepository.save(photo); }
19
20      public void deletePhoto(Photo photo) { photoRepository.delete(photo); }
21
22      public List<Photo> findAll() { return photoRepository.findAll(); }
23
24      public List<Photo> getByPublique(boolean publique) { return photoRepository.findByPublique(publique); }
25
26      public List<Photo> getByUserId(Long id) { return photoRepository.findById(id); }
27
28      public Optional<Photo> getById(Long id) { return photoRepository.findById(id); }
29
30      public List<Photo> getByAlbumId(Long id) { return photoRepository.findByAlbumId(id); }
31
32      public List<Photo> getByCategoryId(Long id) {
33          return photoRepository.findByCategoryId(id);
34      }
35  }
```

## IV.7.1 Couche de présentation : Controller

Dans le modèle MVC, un **contrôleur gère les interactions entre l'utilisateur et le système**. Dans le cadre d'une application Web, les interactions avec le serveur correspondent aux requêtes HTTP émises et reçues par le navigateur client.



Un contrôleur est une classe Java portant l'annotation **@Controller**. Pour que le contrôleur soit appelé lors du traitement d'une requête, il suffit d'ajouter l'annotation **@RequestMapping** sur une méthode publique de la classe en précisant la méthode HTTP concernée et le chemin d'URI (à partir du contexte de déploiement de l'application) pris en charge par la méthode.

Dans le cas de cette application, les annotations **@GetMapping** et **@PostMapping** sont utilisées à la place de **@RequestMapping**.

## IV.7.1 Couche de présentation : Controller

### Exemple de contrôleur : PhotoController

méthode addPhoto @GetMapping

```
class controller/PhotoController

35 @Controller
36 public class PhotoController {
37
38     @Autowired
39     private UserServices userServices;
40
41     @Autowired
42     private PhotoServices photoServices;
43
44     @Autowired
45     private AlbumServices albumServices;
46
47     @Autowired
48     private CategoryServices categoryServices;
```

```
52 @GetMapping("/add-photo")
53 public String addPhoto(Model model) {
54
55     String username = ((UserLogin) SecurityContextHolder.getContext().
56                         getAuthentication().getPrincipal()).getUsername();
57
58     if (username != null) {
59         User user = new User();
60         user = userServices.findByEmail(username);
61
62         List<Album> myAlbums = albumServices.getAlbumsByUserId(user.getId());
63         model.addAttribute( attributeName: "myAlbums", myAlbums);
64     }
65
66     List<Category> categories = categoryServices.findAll();
67
68     model.addAttribute( attributeName: "categories", categories);
69
70     return "photo/addPhoto";
71 }
```

## IV.7.1 Couche de présentation : Controller

### Exemple de contrôleur : PhotoController

méthode addPhoto @PostMapping

```
72 @PostMapping("/add-photo")
73 public String addPhoto(@Validated Photo photo, BindingResult bindingResult,
74     @RequestParam("id") Optional<Long> id,
75     @RequestParam("fileImage") MultipartFile multipartFile) throws IOException {
76
77     String fileName = StringUtils.cleanPath(multipartFile.getOriginalFilename());
78     photo.setFileName(fileName);
79
80     if (bindingResult.hasErrors()) {
81         System.out.println(bindingResult.hasErrors());
82         System.out.println(bindingResult.getFieldError());
83         return "/photo/addPhoto";
84     }
85
86     String username = ((UserLogin) SecurityContextHolder.getContext().getAuthentication()
87         .getPrincipal()).getUsername();
88
89     if (username != null) {
90         User user = new User();
91         user = userServices.findByEmail(username);
92
93         photo.setUser(user);
94     }
95
96     Photo savedPhoto;
97
98     savedPhoto = photoServices.createPhoto(photo);
99
100    String uploadDir = "src/main/resources/static/photos-files/" + savedPhoto.getId();
101
102    Path uploadPath = Paths.get(uploadDir);
103
104    if (!Files.exists(uploadPath)) {
105        Files.createDirectories(uploadPath);
106    }
107
108    try (InputStream inputStream = multipartFile.getInputStream()) {
109        Path filePath = uploadPath.resolve(fileName);
110        Files.copy(inputStream, filePath, StandardCopyOption.REPLACE_EXISTING);
111    } catch (IOException e) {
112        throw new IOException("Impossible de sauvegarder le fichier : " + fileName, e);
113    }
114
115    return "redirect:/my-photos";
116}
117
118
119    return "home";
120}
```

## IV.7.2 Couche de présentation : templates thymeleaf

fichiers template

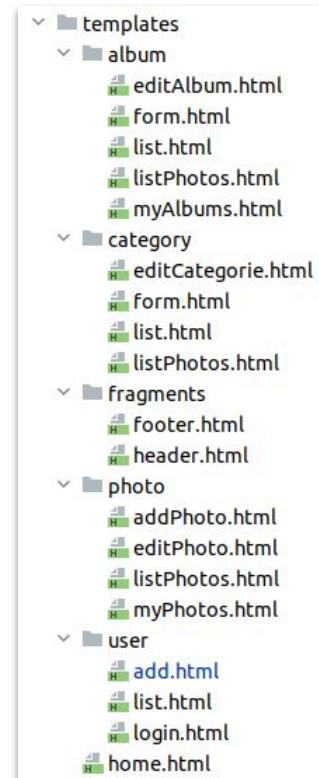
Thymeleaf est un moteur de rendu de document pour les vues qui sont écrites en HTML et CSS.

Pour configurer Thymeleaf, il faut ajouter la dépendance dans le fichier **pom.xml** :

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

Les attributs Thymeleaf sont ajoutés dans des balises HTML pour être interprétés par le serveur permettant l'affichage des données.

Le contenu des attributs Thymeleaf sera une expression qui sera interprétée à l'exécution. Une syntaxe permet d'avoir des expressions à évaluer, de sélections à évaluer, de messages ou d'URLs.



## IV.7.2 Couche de présentation : templates et thymeleaf

### Exemple de fichiers de template : liste de tous les albums

Dans cet exemple, l'**attribut th:if** permet de vérifier si l'objet album est vide ou non.

L'**attribut th:each** permet de parcourir la liste des albums et d'afficher les données de la table album via l'**attribut th:text**.

Une expression ternaire permet de vérifier s'il existe un user dans l'objet album.

```
extrait album/list.html
<div class=p-3>
    <h1>Liste des albums</h1>
    <div class="btn btn-outline-warning" th:if="${albums.isEmpty()}">
        Pas d'album
    </div>
    <div th:if="${!albums.isEmpty()}">
        <table class="table">
            <tbody>
                <tr th:each="album:${albums}">
                    <th scope="row" th:text="${album.id}"></th>
                    <td th:text="${album.name}"></td>
                    <td th:text="${album.user} ? ${album.user.firstname} +
                        ' ' + ${album.user.name} : 'Inconnu'"></td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

## IV.7.2 Couche de présentation : templates et thymeleaf

**Exemple de fichiers de template : formulaire d'ajout d'un album.**

Dans cet exemple, l'**attribut th:object** permet de lier le formulaire à l'**objet album**.

L'**attribut th:field** permet de spécifier le chemin d'accès à la valeur du champ, ici **name**.

*extrait album/editAlbum.html*

```
<div class="formulaire p-3">
    <h1>Modifier le nom de l'album</h1>
    <form method="post" th:object="${album}">
        <div class="mb-3">
            <label for="nom" class="form-label">Nom</label>
            <input type="text" class="form-control" id="nom" name="name" th:field="*{name}">
        </div>
        <button type="submit" class="btn btn-primary">Modifier</button>
    </form>
</div>
```

## IV.8.1 Focus : import de fichiers

Cette application repose sur le téléchargement de fichiers.

Dans la classe d'entité Photo, c'est par l'attribut **private String fileName**; que sera enregistré dans la base de données le nom du fichier téléchargé.

Dans la balise form du template addPhoto.html, l'attribut **enctype="multipart/form-data"** indique que ce formulaire contient un fichier à télécharger.

Pour gérer le fichier téléchargé depuis le client, nous devons déclarer **@RequestParam ("fileImage") MultipartFile multipartFile** en paramètre pour la méthode addPhoto du controller Photo.

Ensuite, nous obtenons le nom du fichier téléchargé, le définissons dans le champ filename de l'objet photo , qui est ensuite conservé dans la base de données :

```
String fileName = StringUtils.cleanPath(multipartFile.  
getOriginalFilename());  
photo.setFileName(fileName);
```

extrait de controller/PhotoController.java

```
@PostMapping("/add-photo")  
public String addPhoto(@Validated Photo photo, BindingResult bindingResult,  
@RequestParam("id") Optional<Long> id,  
@RequestParam("fileImage") MultipartFile multipartFile) throws IOException {  
  
    String fileName = StringUtils.cleanPath(multipartFile.getOriginalFilename());  
    photo.setFileName(fileName);  
  
    photo.setUser(user);  
    Photo savedPhoto;  
    savedPhoto = photoServices.createPhoto(photo);  
  
    String uploadDir = "src/main/resources/static/photos-files/" + savedPhoto.getId();  
    Path uploadPath = Paths.get(uploadDir);  
  
    if (!Files.exists(uploadPath)) {  
        Files.createDirectories(uploadPath);  
    }  
    try (InputStream inputStream = multipartFile.getInputStream()) {  
        Path filePath = uploadPath.resolve(fileName);  
        Files.copy(inputStream, filePath, StandardCopyOption.REPLACE_EXISTING);  
    } catch (IOException e) {  
        throw new IOException("Impossible de sauvegarder le fichier : " + fileName, e);  
    }  
  
    return "redirect:/my-photos";  
}
```

## IV.8.1 Focus : import de fichiers

Seul le nom du fichier est enregistré dans la table de la base de données, le fichier téléchargé est stocké dans le système de fichiers.

Ici, le fichier téléchargé est stocké dans le répertoire `src/main/static/photo-files/:photoid`, qui est relatif au répertoire de l'application. Le fichier est stocké dans un répertoire nommé par son Id, il sera donc unique.

Le répertoire est créé s'il n'existe pas et le fichier téléchargé à partir de l'objet `MultipartFile` est enregistré dans un fichier du système de fichiers.

```
Photo savedPhoto;
savedPhoto = photoServices.createPhoto(photo);
String uploadDir = "src/main/resources/static/photos-files/" + savedPhoto.getId();
Path uploadPath = Paths.get(uploadDir);
if (!Files.exists(uploadPath)) {
    Files.createDirectories(uploadPath);
}
try (InputStream inputStream = multipartFile.getInputStream()) {
    Path filePath = uploadPath.resolve(fileName);
    Files.copy(inputStream, filePath, StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) {
    throw new IOException("Impossible de sauvegarder le fichier : " + fileName, e);
}
```

## IV.8.2 Focus : affichage de fichiers importés

Il faut exposer le répertoire contenant les fichiers téléchargés afin que les clients (navigateurs Web) puissent y accéder. Spring Boot est configuré pour autoriser l'accès au répertoire **src/main/resources/static/photos-files** dans le système de fichiers.

```
@Configuration
public class MvcConfig implements WebMvcConfigurer{
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        Path photoUploadDir = Paths.get("src/main/resources/static/photos-files");
        String photoUploadPath = photoUploadDir.toFile().getAbsolutePath();

        registry.addResourceHandler("src/main/resources/static/photos-files/**")
            .addResourceLocations("file://" + photoUploadPath + "/");
    }
}
```

extrait de configuration/MvcConfig.java

Un getter est ajouté dans l'entité Photo permettant d'avoir accès au chemin du fichier :

```
@Transient
public String getFileNameImagePath() {
    if (fileName == null || id == null) return null;
    return "/photos-files/" + id + "/" + fileName;
}
```

Enfin dans la vue, l'image est affichée via un attribut thymeleaf avec la syntaxe du javascript inline de thymeleaf :

```

```

## IV.8.3 Focus : bibliothèque Leaflet

Leaflet est une bibliothèque JavaScript open source permettant de faire des cartes Web interactives.

Il faut avant tout appeler la bibliothèque leafLet :

*extrait de album/editAlbum.html*

```
18 <script src="https://unpkg.com/leaflet@1.8.0/dist/leaflet.js"
19     integrity="sha512-BB3hKbKW0c9Ez/TAwyWxNxeoV9c1v6F1eYiBieIWkpLjauysF18NzgR1MBNBXf8/KABdLkX68nAhlwDFLGPCQ=="
20     crossorigin=""></script>
21 <script type='text/javascript' th:src="@{js/initLeaflet.js}"></script>
```

Puis définir dans le template une div avec une hauteur dans laquelle se placera la carte :

*extrait de css/style.css*

*extrait de album/editAlbum.html*

```
2 <div id="map"></div>
```

```
2 #map {
3     height:calc(100vh - 56px);
4     z-index: 1;
5 }
```

## IV.8.3 Focus : bibliothèque Leaflet

Il faut ensuite **initialiser la carte** avec des paramètres :

*extrait de js/initLeaflet.js*

```
1 // On initialise la latitude et la longitude de Paris (centre de la carte)
2 var lat = 48.852969;
3 var lon = 2.349903;
4 var macarte = null;
5 var markerClusters; // Sera à stocker les groupes de marqueurs
6
7 // Fonction d'initialisation de la carte
8 function initMap() {
9     var markers = []; // Nous initialisons la liste des marqueurs
10    // Créer l'objet "macarte" et l'insérer dans l'élément HTML qui a l'ID "map"
11    macarte = L.map('map').setView([lat, lon], 11);
12    markerClusters = L.markerClusterGroup(); // Nous initialisons les groupes de marqueurs
13    // Leaflet ne récupère pas les cartes (tiles) sur un serveur par défaut. Nous devons
14    // lui préciser où nous souhaitons les récupérer. Ici, openstreetmap.fr
15    L.tileLayer('https://s.tile.openstreetmap.fr/osmfr/{z}/{x}/{y}.png', {
16        // Il est toujours bien de laisser le lien vers la source des données
17        attribution: 'données © <a href="//osm.org/copyright">OpenStreetMap</a>/ODbL'
18        + '- rendu <a href="//openstreetmap.fr">OSM France</a>',
19        minZoom: 1,
20        maxZoom: 20
21    }).addTo(macarte);
```

La variable **markerClusters** permet de regrouper des marqueurs quand il y en a plusieurs dans une zone géographique rapprochée.

**L.titleLayer** permet de charger la carte voulue, ici la version française (osmfr) de la carte OpenStreetMap.

Les variables z, x et y définissent le niveau de zoom et la position.

## IV.8.3 Focus : bibliothèque Leaflet

L'affichage des photos se fait par une popup après avoir cliqué sur un marqueur :

```
22 // Nous parcourons la liste des photos
23 /*<![CDATA[*/
24
25     /*[# th:each="photo : ${photos}"]*/
26     /*[+
27         var latitude = [[${photo.latitude}]]
28     */+
29     /*[+
30         var longitude = [[${photo.longitude}]]
31     */+
32         var marker = L.marker([latitude,longitude]);
33         // Nous ajoutons la popup.
34     /*[+
35         var infosPhoto = '<h4>' + [[${photo.title}]] +
36             '</h4><p><img class="thumbnail" src=' +
37             [[${photo.FileNameImagePath}]] + "'/></p><p>Photographe : ' +
38             [[${photo.user} ? ${photo.user.firstname} + ' ' + ${photo.user.name} : 'Inconnu']] +
39             '</p><p>Date : ' + [[${#dates.format(photo.date, 'dd MMMM YYYY')} ]] + '</p><p>' +
40             [[${photo.description}]] + '</p><p>Album : ' +
41             [[${photo.album} ? ${photo.album.name} : 'Inconnu']] + '</p>'
42     */+
43         marker.bindPopup(infosPhoto);
44         markerClusters.addLayer(marker); // Nous ajoutons le marqueur aux groupes
45         markers.push(marker); // Nous ajoutons le marqueur à la liste des marqueurs
46
47     /*[/*]*/
48 /*]]>*/
```

Dans cet extrait de code se trouve la syntaxe pour afficher les attributs thymeleaf dans un fichier JavaScript.

Toutes les informations de l'objet photo sont ajoutées dans la **variable infosPhoto**.

## IV.9 Sécurité : Spring Security

Dans le **package Service** : création de la classe UserLoginDetailsService qui implémente UserDetailsService.

*extrait de services/UserDetailsService.java*

```
public class UserLoginDetailsService implements UserDetailsService{  
  
    @Autowired  
    private UserRepository userRepository;  
  
    @Override  
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
  
        User user = userRepository.findByEmail(username);  
        if (user == null) {  
            throw new UsernameNotFoundException("User not found");  
        }  
        return new UserLogin(user);  
    }  
}
```

Pour sécuriser l'application, il faut récupérer le username dans la base de données.

## IV.9 Sécurité : Spring Security

Dans le package configuration :  
création de la classe SecurityConfig

Dans la méthode **configure()** il est possible de définir la **politique de sécurité** de l'application : l'accès est autorisé à tous les visiteurs sur les pages d'accueil et de connexion, mais seulement aux utilisateurs connectés sur les autres pages.

En cas de **non authentification**, les pages qui en demandent une renverront sur la **page de login**

Sur la page de login, **une erreur** de l'adresse email ou du mot de passe renverra l'url "**/login?error=true**".

extrait de configuration/SecurityConfig.java

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .csrf().disable()  
        .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry  
        .antMatchers( ...antPatterns: "/users").authenticated()  
        .antMatchers( ...antPatterns: "/add-album").authenticated()  
        .antMatchers( ...antPatterns: "/my-albums").authenticated()  
        .antMatchers( ...antPatterns: "/add-photo").authenticated()  
        .antMatchers( ...antPatterns: "/my-photos").authenticated()  
        .antMatchers( ...antPatterns: "/my-photos-list").authenticated()  
        .antMatchers( ...antPatterns: "/list-users").authenticated()  
        .antMatchers( ...antPatterns: "/list-albums").authenticated()  
        .antMatchers( ...antPatterns: "/list-photos").authenticated()  
        .antMatchers( ...antPatterns: "/add-category").authenticated()  
        .antMatchers( ...antPatterns: "/list-categories").authenticated()  
        .anyRequest().permitAll()  
        .and()  
        .HttpSecurity  
        .formLogin() FormLoginConfigurer<HttpSecurity>  
        .LoginPage("/login")  
            .usernameParameter("email")  
            .defaultSuccessUrl("/").failureUrl( authenticationFailureUrl: "/login?error=true").permitAll()  
        .and()  
        .HttpSecurity  
        .logout() LogoutConfigurer<HttpSecurity>  
        .logoutRequestMatcher(new AntPathRequestMatcher( pattern: "/logout"))  
        .logoutSuccessUrl("/")  
        .permitAll();  
}
```

## IV.9 Sécurité : Spring Security

### Spring Security et Thymeleaf

Il faut tout d'abord ajouter un module d'intégration entre Thymeleaf et Spring Security

Dans le template Thymeleaf l'attribut

`sec:authorize="isAuthenticated()"` placé dans une balise HTML permet de n'afficher le contenu de cette balise seulement si l'utilisateur est authentifié.

Au contraire, l'attribut

`sec:authorize="isAnonymous()"` placé dans une balise HTML permet d'afficher le contenu si l'utilisateur n'est pas authentifié.

extrait de pom.xml

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
```

extrait de fragments/header.html

```
<div class="navbar-text">
    <span th:text="${session.name}">John Doe</span>
    <div class="d-flex" sec:authorize="isAuthenticated()">
        <div class="dropdown mx-1" ...>
            <a class="btn btn-danger" th:href="@{/logout}">Déconnexion</a>
            <!-- ... -->
        </div>
        <span sec:authorize="isAnonymous()">
            <a class="btn btn-outline-light" th:href="@{/login}" role="button">Connexion</a>
            <a class="btn btn-light" th:href="@{/signup}" role="button">Inscription</a>
        </span>
    </div>
</div>
```

## IV.9 Sécurité : Spring Security

### Dans le package controller : vérification de l'authentification

Pour qu'un utilisateur puisse accéder à ses albums, il faut vérifier qu'il soit connecté.

Dans la méthode `myAlbum` de la classe `Album`, `String username = ((UserLogin) SecurityContextHolder.getContext().getAuthentication().getPrincipal()).getUsername()`; permet de récupérer le username de l'utilisateur connecté.

```
// liste de mes albums
@GetMapping("/my-albums")
public String myAlbums(Model model) {
    String username = ((UserLogin) SecurityContextHolder.getContext().getAuthentication().getPrincipal()).getUsername();

    User user = new User();
    user = userRepository.findByEmail(username);

    List<Album> myAlbums = albumServices.getAlbumsByUserId(user.getId());

    model.addAttribute(attributeName: "myAlbums", myAlbums);

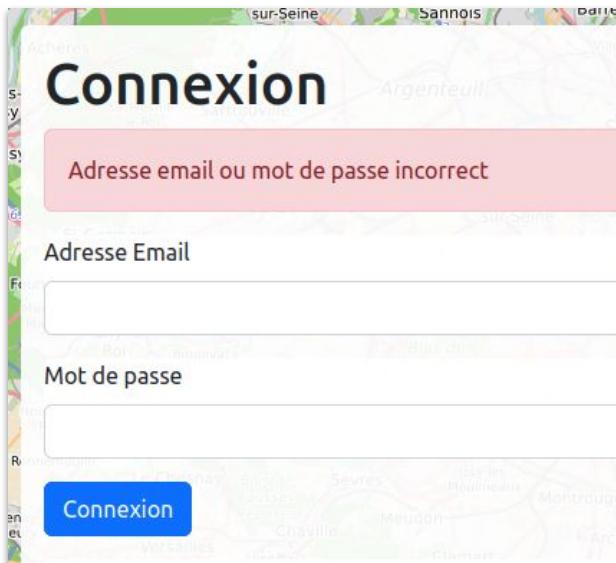
    return "album/myAlbums";
}
```

*extrait de controller/album.java*

## IV.9 Sécurité : Spring Security

### Page de login

Dans la vue, si l'url contient le **paramètre "error=true"**, un message d'erreur s'affiche.



capture d'écran de la page

<http://localhost:8081/geolocPhotos/login?error=true>

```
<form method="post">
    <div class="mb-3" th:if="${param.error}">
        <p class="alert alert-danger" role="alert">Adresse email ou mot de passe incorrect</p>
    </div>
    <div class="mb-3">
        <label for="email" class="form-label">Adresse Email</label>
        <input type="email" class="form-control" id="email" name="email">
    </div>
    <div class="mb-3">
        <label for="Password1" class="form-label">Mot de passe</label>
        <input type="password" class="form-control" id="Password1" name="password">
    </div>
    <button type="submit" class="btn btn-primary">Connexion</button>
</form>
```

extrait de user/login.html

## IV.9 Sécurité : Spring Security

### Page d'inscription

Dans l'entité User, l'annotation `@Email` permet d'imposer que l'adresse email soit valide et l'annotation `@Pattern` permet d'imposer un format au mot de passe.

The screenshot shows a registration form titled "Inscription". It includes fields for Nom (Name), Prenom (First Name), Email (Email), and Mot de passe (Password). The Email field contains "jean.martin" and has a red error message: "Adresse email non valide". The Password field has a red error message: "Le mot de passe doit contenir des minuscules, majuscules, des chiffres et des caractères spéciaux la longueur doit être comprise entre 8 et 100". Below the form is a map with coordinates: longitude: 2.3950195 and latitude: 48.297813. A "Submit" button is at the bottom.

capture d'écran de la page  
<http://localhost:8081/geolocPhotos/signup>

```
@Email(message = "Adresse email non valide")
@NotEmpty(message = "Adresse email obligatoire")
@Column(name="email", length=255, nullable=false, unique=true)
private String email;

@Transient
private String oldEmail;

@NotNull(message = "le mot de passe ne peut être vide")
@Pattern(regexp = "^(a-zA-Z_0-9!@#$%^&){8,100}$",
message = "Le mot de passe doit contenir des minuscules, majuscules, " +
"des chiffres et des caractères spéciaux !@#$%^&")
@Length(min = 8, max = 100)
private String password;
```

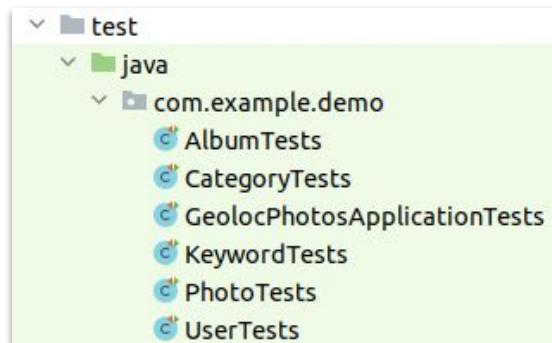
extrait de model/User

## IV.10 Tests unitaires

### Tests des méthodes sur l'objet User

Les tests unitaires sont réalisés avec le framework **JUnit**.

Ils sont répartis dans différentes classes et sont tous rassemblés dans un **package de tests**.



fichiers de tests

## IV.10 Tests unitaires

Ici sont présentés les tests unitaires sur les méthodes CRUD de l'utilisateur.

Dans cette première partie, deux tests sont réalisés :

- un test pour **créer un utilisateur**. On crée un utilisateur avec ses attributs. Le test consiste à vérifier si l'objet User existe.
- un test pour **vérifier qu'un utilisateur existe**. On cherche un utilisateur par son adresse email (unique) dont on sait au préalable qu'elle est bien présente dans la base de données. Le test consiste à vérifier si l'objet existe.

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class UserTests {

    @Autowired
    private UserRepository userRepository;

    @Test
    @Rollback(false)
    @Order(1)
    public void testCreateUser() {
        User user = new User(name: "Dupont", firstname: "Jean",
                             email: "email@test.fr", password: "1234", (float) 45.34,
                             (float) -10.12, role: true);
        User savedUser = userRepository.save(user);

        assertNotNull(savedUser);
    }

    @Test
    @Rollback(false)
    @Order(2)
    public void testEmailExists() {
        String email = "email@test.fr";
        User user = userRepository.findByEmail(email);

        assertNotNull(user);
    }
}
```

extrait de src/test/java/UserTests.java

## IV.10 Tests unitaires

Dans cette deuxième partie, deux tests sont réalisés :

- un test pour **vérifier qu'un utilisateur n'existe pas**. On cherche un utilisateur par son adresse email (unique) dont on sait au préalable qu'elle n'est pas présente dans la base de données. Le test consiste à vérifier que l'objet n'existe pas.
- un test pour **mettre à jour un utilisateur**. On récupère un utilisateur par son adresse email, puis on modifie son nom. On crée ensuite une seconde instance de l'objet avec la même adresse email. Le test consiste à vérifier que le nom des deux instances User sont les mêmes.

```
@Test  
@Rollback(false)  
@Order(3)  
public void testEmailDoesntExist() {  
    String email = "emaildoesntexist@test.fr";  
    User user = userRepository.findByEmail(email);  
  
    assertNull(user);  
}  
  
@Test  
@Rollback(false)  
@Order(5)  
public void testUpdateUser() {  
    String email = "email@test.fr";  
    User user = userRepository.findByEmail(email);  
    user.setName("Martin");  
  
    userRepository.save(user);  
    User updatedUser = userRepository.findByEmail(email);  
  
    assertEquals(user.getName(), updatedUser.getName());  
}
```

extrait de src/test/java/UserTests.java

## IV.10 Tests unitaires

Dans cette troisième partie, deux tests sont réalisés :

- un test pour vérifier qu'il y a des utilisateurs. On récupère la liste de tous les utilisateurs. Le test consiste à vérifier que le nombre d'objets dans la liste est supérieur à zéro.
- un test pour supprimer un utilisateur. On récupère un utilisateur par son adresse email, puis on modifie son nom, puis on supprime l'instance. On récupère le même utilisateur dans une nouvelle instance. Le test consiste à vérifier que la première instance existe et que la seconde n'existe pas.

```
@Test  
@Rollback(false)  
@Order(4)  
public void testListUsers() {  
    List<User> users = userRepository.findAll();  
  
    assertTrue(condition: users.size() > 0);  
}  
  
@Test  
@Rollback(false)  
@Order(6)  
public void testDeleteUser() {  
    String email = "email@test.fr";  
  
    User userExists = userRepository.findByEmail(email);  
  
    userRepository.deleteById(userExists.getId());  
  
    User userDoesntExist = userRepository.findByEmail(email);  
  
    assertNotNull(userExists);  
    assertNull(userDoesntExist);  
}
```

extrait de src/test/java/UserTests.java

# AFFICHAGE DANS LE NAVIGATEUR

# V.1 Page d'accueil

Accueil À propos

Connexion

Inscription

71

## V.2 Page d'inscription d'un nouvel utilisateur

Connexion | Inscription

Accueil | À propos

**Inscription**

Nom  
Jean

Prenom  
Martin

Email  
jean.martin

Mot de passe  
\*\*\*\*\*

Votre mot de passe doit contenir entre 8 et 100 caractères avec des minuscules, majuscules, chiffres, caractères spéciaux

Cliquer sur la carte pour définir votre lieu par défaut

longitude : 2.5048828125000004

latitude : 46.99524110694596

Submit

72

## V.3 Page d'inscription d'un nouvel utilisateur avec messages d'erreurs

Accueil À propos Connexion Inscription

The map displays major cities across Europe, including London, Paris, Berlin, and Rome, with a focus on the British Isles and the Iberian Peninsula.

**Inscription**

Nom: Jean

Prenom: Martin

Email: jean.martin

Adresse email non valide

Mot de passe:

Votre mot de passe doit contenir entre 8 et 100 caractères avec des minuscules, majuscules, chiffres, caractères spéciaux

Le mot de passe doit contenir des minuscules, majuscules, des chiffres et des caractères spéciaux  
la longueur doit être comprise entre 8 et 100

Cliquer sur la carte pour définir votre lieu par défaut

longitude : 2.3950195

latitude : 48.297813

Submit

# V.4 Page de connexion

The screenshot shows a map-based login interface. At the top left is a navigation bar with "Accueil" and "À propos". At the top right are "Connexion" and "Inscription" buttons. The main area features a large map of the Paris region, specifically the Yvelines and Seine-Saint-Denis departments, with numerous towns and landmarks labeled. Overlaid on the map is a central login form. It includes fields for "Adresse Email" containing "admin@geolphotos.fr", "Mot de passe" (with four asterisks), and a "Connexion" button. Below the password field is a link "Pas encore de compte ? [inscrivez-vous !](#)".

# V.5 Page d'ajout d'une photo

Accueil Photos ▾ Album ▾ À propos

Amselle Guillaume ▾ Déconnexion

Pour ajouter une photo, cliquer sur la carte

Télécharger votre photo

Parcourir... 20200219215039-d04e3298-la.jpg



Titre

Notre-Dame

Date

02 / 11 / 2022

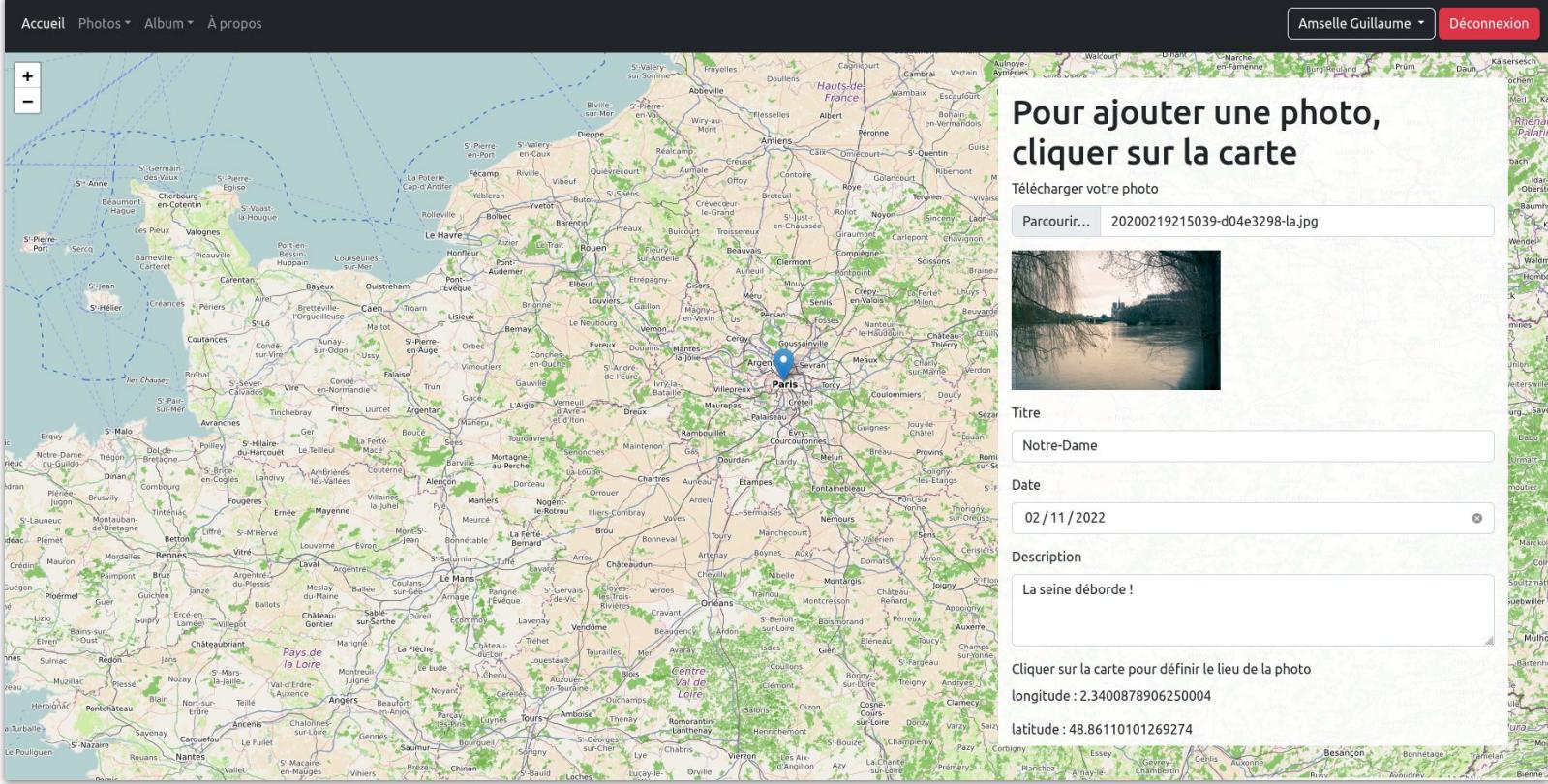
Description

La seine déborde !

Cliquer sur la carte pour définir le lieu de la photo

longitude : 2.3400878906250004

latitude : 48.86110101269274



# V.6 Page d'ajout d'un album

The screenshot shows a map of the Paris region with numerous towns and landmarks labeled. Overlaid on the map is a form for creating a new album. The form includes:

- A header bar with "Accueil", "Photos", "Album", "À propos", "Amselle Guillaume" (dropdown), and "Déconnexion".
- A title field: "Nouvel album".
- A subtitle field: "nom de l'album".
- A text input field: "Italie été 2022".
- A blue "Ajouter" button.

## V.7 Page liste des albums d'un utilisateur

The screenshot shows a user interface for managing photo albums. At the top, there is a navigation bar with links for Accueil, Photos, Album, and À propos. On the right side of the header, there are buttons for Amselle Guillaume (dropdown menu) and Déconnexion. Below the header, a map of the Paris region is displayed, with two specific albums overlaid: 'Paris' and 'Italie été 2022'. Each album has a thumbnail image, a title, and two buttons: 'Modifier' (blue) and 'Supprimer' (red). The 'Paris' album is dated 2022-07-16 and the 'Italie été 2022' is dated 2022-07-17.

Album	Date	Action
Paris	2022-07-16	Modifier Supprimer
Italie été 2022	2022-07-17	Modifier Supprimer

# CONCLUSION

# Conclusion

Dans l'état actuel, cette application est partiellement fonctionnelle. Il est possible de publier des photos et de les visualiser.

Cependant, compte tenu des conditions de sa réalisation, beaucoup reste encore à faire. Parmi les évolutions à apporter, je pourrais, par exemple :

- Implémenter les mots-clés et les commentaires ;
- Ajouter des fonctionnalités comme les favoris ;
- Ajouter l'interface « classique » ;
- Récupérer directement les données EXIF des photos pour afficher les informations de la photo ;
- Afficher les images en pleine taille.
- Reprendre le développement pour transformer l'application en une API Restful.

La réalisation de ce projet aura été l'occasion d'une part d'approfondir ma connaissance du framework Spring Boot vu pendant la formation et d'autre part de découvrir la bibliothèque Javascript LeafLet. Ce projet m'aura aussi permis d'approfondir ma compréhension du design pattern MVC, et plus particulièrement celui de Spring Boot, appris pendant la formation.

Je terminerais par un mot sur mon stage en entreprise qui ne m'a pas permis de présenter un projet pour la certification du CDA. Ce stage m'a permis néanmoins de découvrir le principe d'une API REST dans le cadre du framework Spring Boot et d'avoir un aperçu du framework ReactJS.