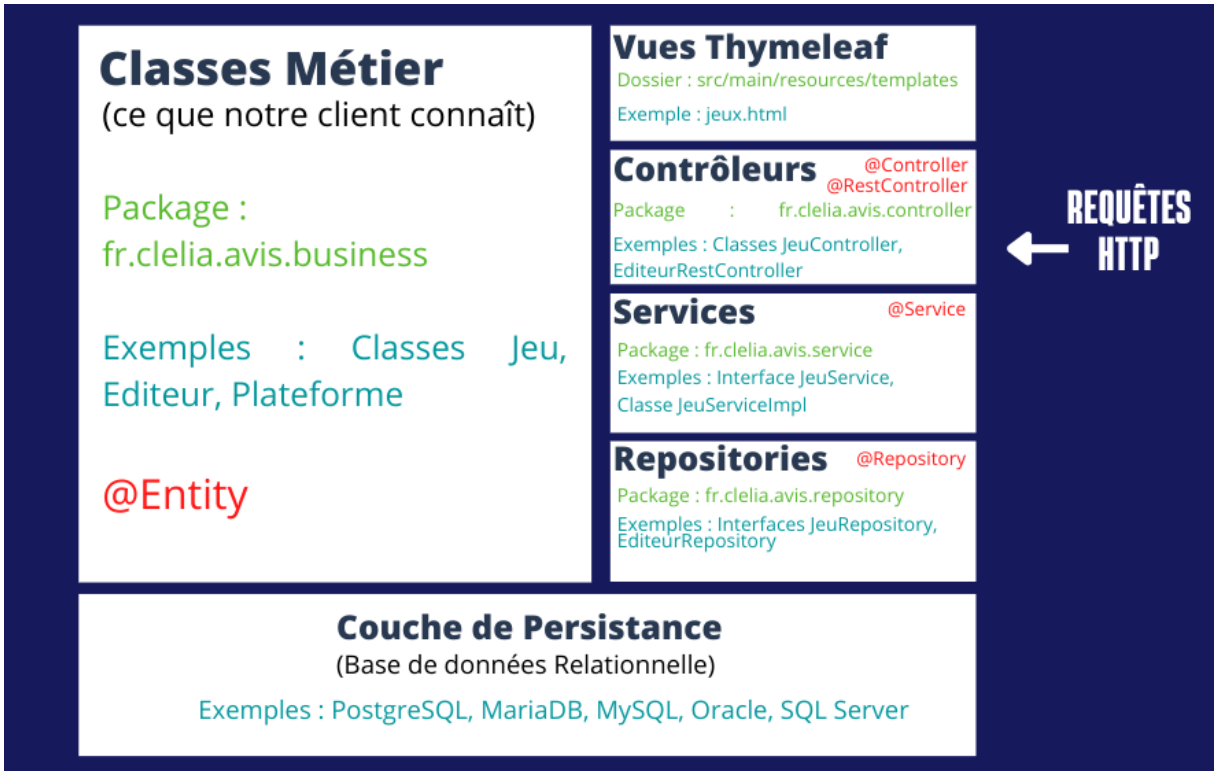
Spring

Cook Book

Auteur du document : François-Xavier COTE (fxcote@clelia.fr)
Version du document : 1.73 en date du 08/04/2025

Ce Cook Book décrit les étapes pour créer un projet Spring :
- embarquant un serveur Tomcat qui écoute sur le port 8080
- embarquant une base de données H2 en mémoire nommée avis
- utilisant des vues Thymeleaf et Bootstrap pour le front
- offrant une API REST documentée avec Swagger
- qualifié de monolithique et basé sur le modèle en 5 couches comme suit :



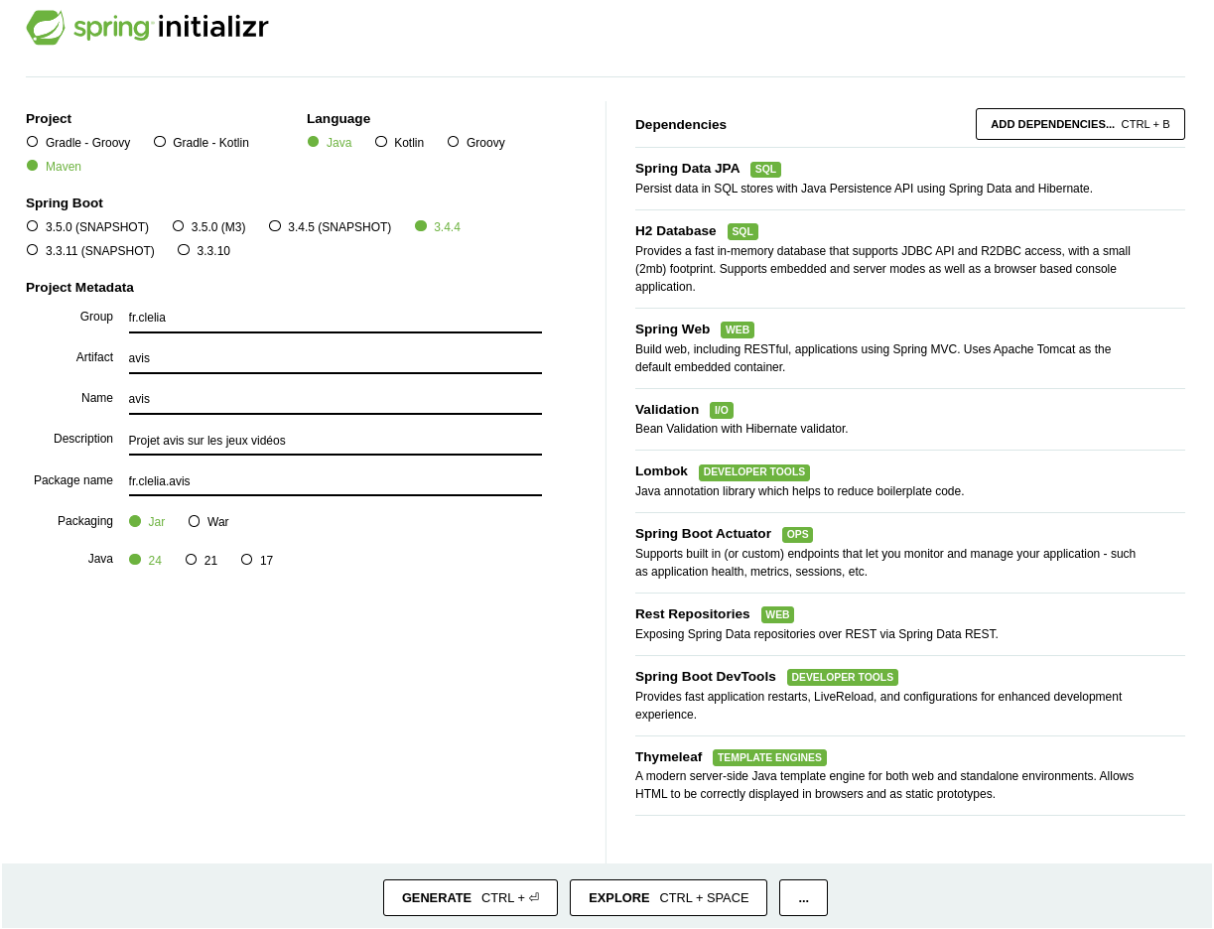
Avec ce modèle en 5 couches :

- les dépendances entre les couches se matérialisent par des directives d'import
- la programmation par contrat est mise en œuvre pour réduire le couplage : la communication entre les couches se base sur la notion d'interface.

Exemple le contrôleur JeuController a besoin d'un service JeuService, il déclare cette dépendance en faisant référence à l'interface et non à l'implémentation du service (JeuServiceImpl) :

```
@Controller
public class JeuController {

   private final JeuService jeuService;

}
```

1) Générer le projet Maven à l'aide de Spring Initializr



Cliquez ici pour obtenir la même configuration :

https://start.spring.io/#!type=maven-project&language=java&platformVersion=3.4.4&packaging=jar&jvmVersion=21&groupId=fr.clelia&artifactId=avis&name=avis&description=Projet%20avis%20sur%20les%20jeux%20vid%C3%A9os&packageName=fr.clelia.avis&dependencies=data-jpa,h2,web,validation,lombok,actuator,data-rest,devtools,thymeleaf

Ce site évolue aussi vite que Spring Boot :
https://github.com/spring-projects/spring-boot/milestones

En sélectionnant Maven, le nouveau projet aura pour parent le projet spring-boot-starter-parent. Ce dernier a également un projet parent : spring-boot-dependencies qui définit la version de chaque dépendance liée à Spring :

```
<junit.version>4.13.2</junit.version>
<junit-jupiter.version>5.11.4</junit-jupiter.version>
<kafka.version>3.8.1</kafka.version>
```

Alternativement, pour générer le projet Maven :
- avec Eclipse et le plugin Spring Tool Suite : File / New / Spring Boot / Spring Starter Project
- avec IntelliJ et les plugins Spring et Spring Boot : File / New / Project / Spring Initializr

Si le projet utilise une base de données, un driver JDBC adéquat est requis (sur la capture de la deuxième page, il s'agit de H2 Driver).

Pour que l'application embarque un serveur Tomcat, la dépendance Spring Web sera ajoutée.

Si vous souhaitez que l'application redémarre dès qu'elle détecte un changement dans les fichiers du projet, la dépendance Spring Boot DevTools est faite pour vous.

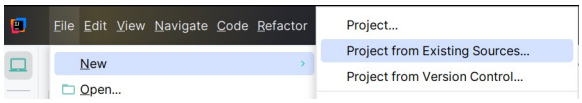
Afin d'obtenir des classes persistantes, autrement dit annotées avec @Entity de JPA (Jakarta Persistance API : https://jakarta.ee/specifications/persistence/3.1/), il faut ajouter la dépendance Spring Data JPA.

Les annotations de validation (@NotNull, @NotBlank, @Size) sont regroupées dans la dépendance "Validation" liée à la spécification de Jakarta
https://jakarta.ee/specifications/bean-validation/

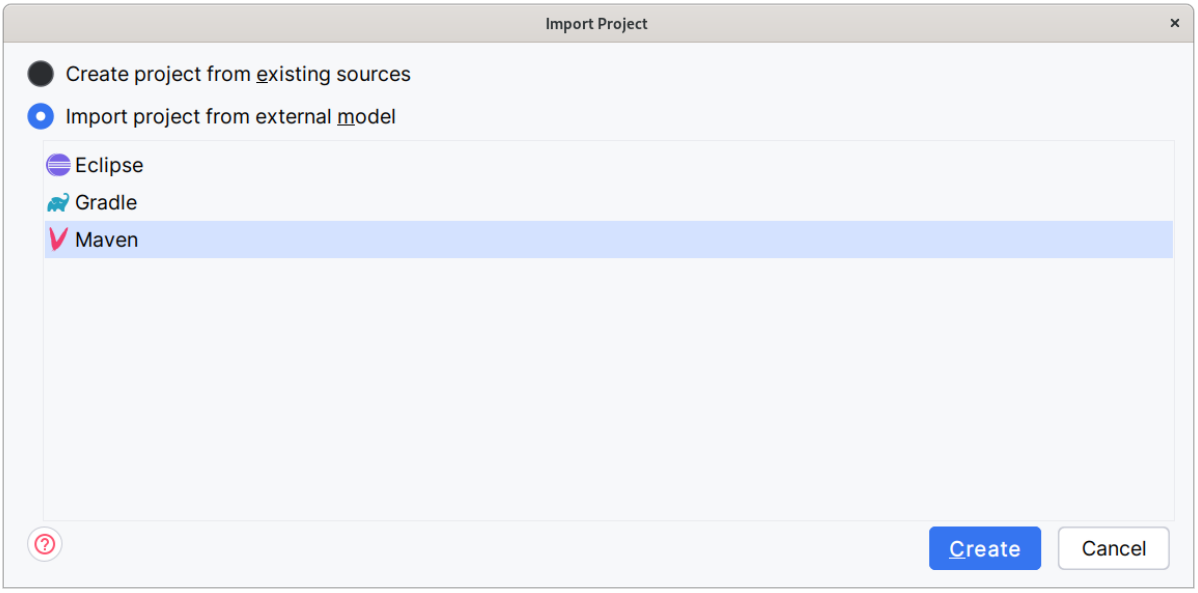
2) Importer le projet dans votre IDE préféré

2.1) S'il s'agit d'Eclipse : décompresser le fichier zip dans le workspace d'Eclipse puis importer le projet Maven en utilisant l'assistant : File / Import / Existing Maven projects

2.2) S'il s'agit d'IntelliJ, décompresser le fichier zip puis ouvrir le projet Maven en utilisant l'assistant File / New / Project from Exisiting Sources :



Préciser le dossier obtenu par décompression puis choisir Maven comme "external model" :



Il reste ensuite à cliquer sur le bouton “Trust project” puis “New Window”.

En cas d’erreur, supprimer le dossier .idea et/ou les fichier .iml et rouvrir de nouveau le projet.

Pour assurer le bon fonctionnement de Spring Boot DevTools dans IntelliJ, vérifiez que les deux options suivantes sont cochées :
- Dans Build, Execution, Deployment | Compiler : “Build project automatically”
- Dans Advanced Settings | Compiler : “Allow auto-make to start even if developed application is currently running”.

2.3) S’il s’agit de VS Code : décompresser le fichier zip et dans VS Code aller dans le menu File / Open Folder puis indiquer le dossier obtenu par décompression

3) Configurer la journalisation

Ajouter les lignes suivantes dans le fichier de configuration de l’application Spring : src/main/resources/application.properties :

```
logging.level.root=WARN
logging.level.org.springframework=WARN
logging.file.name=log/avis_log
logging.pattern.console= %d %p %c{1.} [%t] %m%n
```

Les niveaux de log sont : `FATAL, ERROR, WARN, INFO, DEBUG, TRACE`

Voici le lien vers la document de Log4J concernant les patterns :
https://logging.apache.org/log4j/2.x/manual/layouts.html

Par défaut Spring crée un nouveau fichier pour chaque jour ou dès que le fichier actuel de log atteint 10.5 Mo.

Exemple d'entrée dans le fichier de journalisation :

```
2024-01-07 08:32:18.382  WARN 12698 --- [restartedMain]
JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is
enabled by default. Therefore, database queries may be performed during
view rendering. Explicitly configure spring.jpa.open-in-view to disable
this warning
```

Les logs en couleur nécessitent une nouvelle ligne dans le fichier de configuration :

```
spring.output.ansi.enabled=ALWAYS
```

Une fois la ligne ci-dessus ajoutée, le pattern pour la console pourra utiliser les couleurs comme suit :

```
logging.pattern.console= %d %yellow(%p) %green(%c{1.}) [%t] %m%n
```

Se référer à la documentation officielle pour bien comprendre comment modifier le fichier application.properties :
https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

4) Configurer Spring Boot Actuator

4.1) Pour exposer les beans présents dans le conteneur de Spring (grâce à Actuator) :

```
management.endpoints.web.base-path=/
management.endpoints.web.exposure.include=beans
```

Ce faisant la liste des beans contenus dans le conteneur de Spring sera accessible par l'URL http://localhost:8080/beans

4.2) Pour exposer aussi des informations sur la santé de l'application Spring :

```
management.endpoints.web.exposure.include=beans,health
management.endpoint.health.group.custom.show-components=always
management.endpoint.health.group.custom.show-details=always
```

4.3) Pour exposer également les propriétés de l'application ainsi que les données sur l'environnement Spring :

```
management.endpoints.web.exposure.include=beans,health,env,configprops
management.endpoint.health.group.custom.show-components=always
management.endpoint.health.group.custom.show-details=always
management.endpoint.env.show-values=always
management.endpoint.configprops.show-values=always
```

5) Configurer la persistance des données

5.1) Ajouter dans le fichier src/main/resources/application.properties les lignes suivantes :

```
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
```

A noter : les autres valeurs de ddl-auto sont : `none`, `validate` , `create`, et `create-drop`.
Avec la valeur validate, Hibernate établit un rapport de différence entre le schéma existant et le schéma créé par le scan des entités
Avec la valeur create-drop, Hibernate supprime toutes les tables lorsque l'application s'arrête ce qui se révèle très utile en phase de développement.

5.2) Pour une base H2 en mémoire, le fichier de configuration doit également inclure :

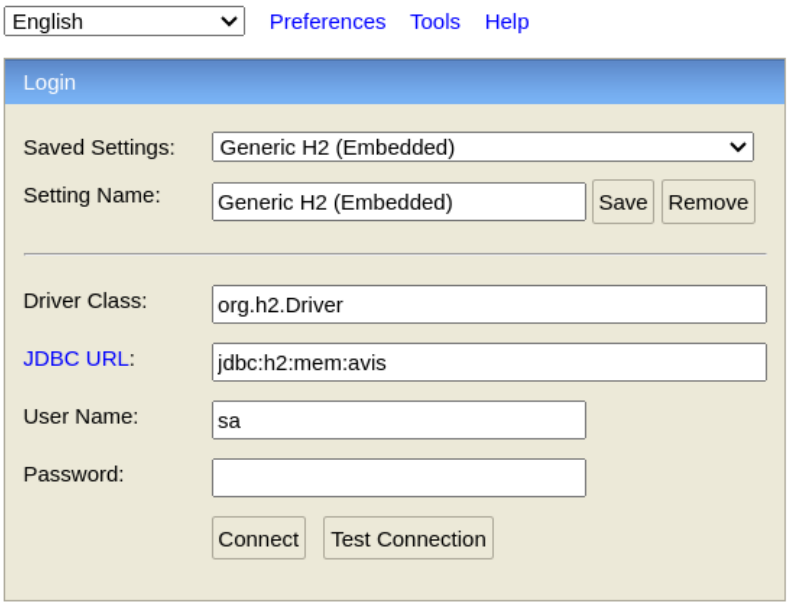
```
spring.datasource.url=jdbc:h2:mem:avis
spring.datasource.driver-class-name=org.h2.Driver
```

Si une base H2 est utilisée en production, vérifier :
- que la balise <scope>test</scope> n'est pas présente dans la dépendance vers H2 du fichier pom.xml.
- que la propriété spring.h2.console.enabled est bien définie à vrai :

```
spring.h2.console.enabled=true
```

Ce faisant, la console H2 sera accessible à cette URL : http://localhost:8080/h2-console

Voici la page de connexion de la console H2 :



Par défaut le champ User Name est sa. Le champ Password peut être laissé vide.

5.3) Pour une base H2 stockée sur le disque dur, le fichier de configuration doit également inclure :

```
spring.datasource.url=jdbc:h2:~/avis
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
```

A partir de la version 6 d'Hibernate, il n'est plus nécessaire de préciser le dialect comme il fallait le faire précédemment :

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

https://docs.jboss.org/hibernate/stable/orm/javadocs/org/hibernate/cfg/JdbcSettings.html#DIALECT

Hibernate va choisir automatiquement le dialect en fonction du driver.

5.4) Pour une base PostgreSQL, le fichier de configuration doit également inclure :


```
spring.datasource.url=jdbc:postgresql://localhost:5432/avis
spring.datasource.username=postgres
spring.datasource.driver-class-name=org.postgresql.Driver
```

5.5) Pour une base MariaDB nommée avis, le fichier de configuration doit inclure :


```
spring.datasource.url=jdbc:mariadb://localhost:3306/avis?useSSL=false
spring.datasource.username=root
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
```

5.6) Pour une base MySQL nommée avis, le fichier de configuration doit également inclure :


```
spring.datasource.url=jdbc:mysql://localhost:3306/avis?useSSL=false
spring.datasource.username=root
```

```
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

6) Écrire/Générer les classes métier

6.1) Soit en **Code First**:

pour chaque classe métier:

6.1.1) ajouter un constructeur vide, sinon on obtient l'exception :
`org.hibernate.InstantiationException: No default constructor for entity: fr.clelia.avis.business.Jeu`

6.1.2) ajouter un accesseur (méthode get) et un mutateur (méthode set) pour chaque attribut privé

6.1.3) une méthode toString(): Spring va se servir de cette méthode pour générer les formulaires HTML utilisant les balises <form:form> et donner à chaque élément du formulaire la bonne valeur par défaut

6.1.4) Annoter les classes business avec les annotations Hibernate (se référer au mémento Annotations)

Exemple:

```
@Entity
public class Jeu {

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Long id;

   private String nom;

    @ManyToOne
   private Editeur editeur;

   public Jeu() {
   }
   …

}
```

Alternativement, Lombok peut être utilisé pour ne plus avoir à écrire les constructeurs, les getters, les setters, la méthode hashCode, equals ainsi que la méthode toString :

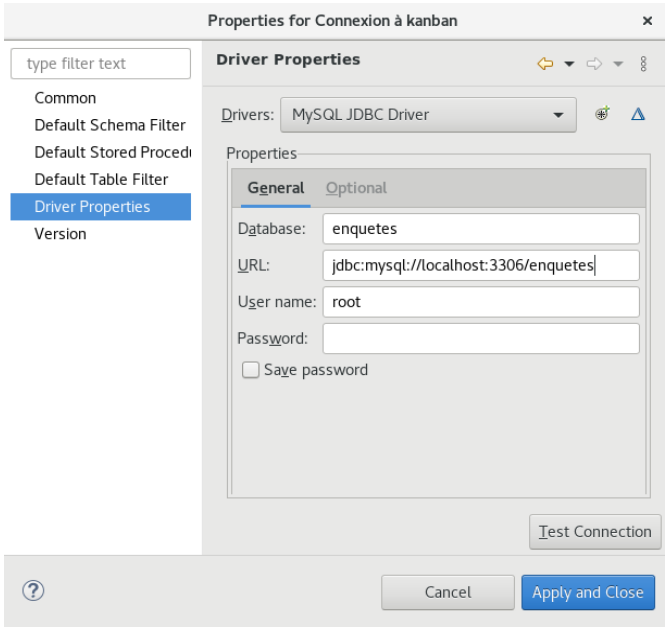
```
@Entity
@NoArgsConstructor
@Data
public class Jeu {

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Long id;

   private String nom;
```

```
    @ManyToOne
    private Editeur editeur;

}
```

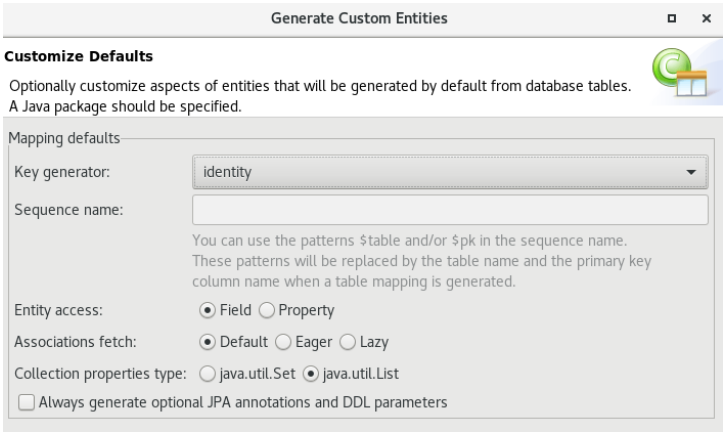
6.2) Soit en **Database First**:

6.2.1) Concevoir les tables avec MySQL Workbench, JMerise ou looping

6.2.2) Ajouter une connexion à la base de données



6.2.3) Cliquer-droit sur le projet: configure / convert to JPA project
6.2.4) JPA est coché, cliquer sur Next
6.2.5) Choisir Generic 2.1 et disable user library puis cliquer sur Finish
6.2.6) Cliquer droit sur le projet JPA Tools / Generate Entities from Tables
6.2.7) Sélectionner toutes les tables, cliquer sur Next 2 fois
6.2.8) Sur la fenêtre « Customize Defaults » chosir identity comme Key generator et comme package le dossier business



Alternativement, les classes métier peuvent être générées en choisissant File / New / JPA Entities from Tables

7) Générer le diagramme de classes métier

StarUML inclut une extension « Java Reverse Engineer » qui génère un diagramme de classes à partir de fichiers Java. Placer le fichier .mdj ainsi qu'une version PNG du diagramme dans un dossier nommé doc.
Alternativement le diagramme de classes métier peut être généré avec le plugin Diagrams d'IntelliJ.

8) Écrire les interfaces repository

Chris EVANS dans son livre Domain-Driven Design définit ainsi une repository :

« A REPOSITORY represents all objects of a certain type as a conceptual set (usually emulated). It acts like a collection, except with more elaborate querying capability. Objects of the appropriate type are added and removed, and the machinery behind the REPOSITORY inserts them or deletes them from the database. This definition gathers a cohesive set of responsibilities for providing access to the roots of AGGREGATES from early life cycle through the end.

Clients request objects from the REPOSITORY using query methods that select objects based on criteria specified by the client, typically the value of certain attributes. The REPOSITORY retrieves the requested object, encapsulating the machinery of database queries and metadata mapping.

REPOSITORIES can implement a variety of queries that select objects based on whatever criteria the client requires. They can also return summary information, such as a count of how many instances meet some criteria. They can even return summary calculations, such as the total across all matching objects of some numerical attribute.»

Dans Eclipse, tout débute dans le menu File / New / Interface. Chaque interface hérite de JpaRepository :

Exemple:

```
public interface JeuRepository extends JpaRepository<Jeu, Long> {}
```

Avec IntelliJ et le plugin JPA Buddy (https://www.jpa-buddy.com/ disponible sur le marketplace d'IntelliJ à partir de la version 2020), les repositories peuvent être générées en choisissant une ou plusieurs classes métier puis clic droit new Spring Data Repository.

Pour installer JPA Buddy sans passer par le marketplace :
- télécharger le fichier zip adapté votre version d'IntelliJ :
https://plugins.jetbrains.com/plugin/15075-jpa-buddy
- Dans settings / Plugins : cliquer sur la roue dentée puis « Install Plugin from Disk… » et indiquer le fichier téléchargé à l'étape précédente

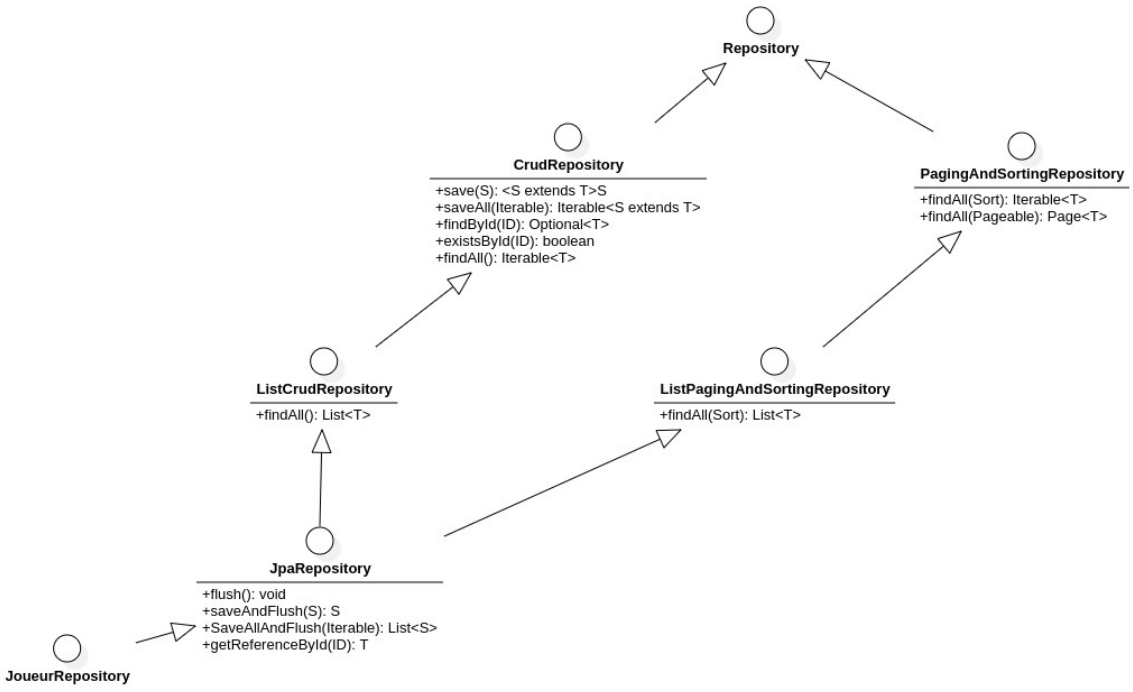
Voici la Javadoc des interfaces Repository de Spring Data JPA :

https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

L'interface JpaRepository hérite de l'interface ListCrudRepository :
https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/ListCrudRepository.html

Les interfaces PagingAndSortingRepository et CrudRepository héritent de l'interface Repository :
https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

Voici un diagramme de classes présentant l'héritage entre les différentes interfaces :



Dans chaque interface du package repository, des méthodes annotées @Query ou des méthodes requêtes peuvent être déclarées. Par défaut l'annotation @Query attend une requête HQL :

```
@Query("""
```

```
            FROM Editeur e
            ORDER BY size(e.jeux) DESC
        """)
List<Editeur> findEditeursSortedByNbJeuxDesc();
```

Se référer à la documentation officielle pour rédiger la requête HQL :
https://docs.jboss.org/hibernate/stable/orm/userguide/html_single/Hibernate_User_Guide.html

L'annotation @Query peut aussi accueillir une requête SQL grâce à l'attribut nativeQuery :

```
@Query(value="SELECT * FROM Jeu WHERE editeur_id=:idEditeur",
nativeQuery=true)
List<Jeu> findByIdEditeur(@Param("idEditeur") Long idEditeur);
```

Une alternative à l'annotation @Query est l'écriture de requêtes par dérivation, en anglais « query-method ». Le nom de la méthode est interprété par Spring Data et traduit en HQL. Exemple :

```
List<Jeu> findByEditeur(Editeur editeur);
```

Les mots clés autorisés dans le nom des méthodes sont résumés sur la table suivante :
https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html

9) Écrire les interfaces puis les classes de service

Annoter chaque classe service avec le stéréotype Spring @Service et demander à Spring l'injection des repositories par l'écriture d'un constructeur ayant en paramètre les objets que Spring doit injecter dans le service.

Exemple:

```
          @Service
          public class JeuServiceImpl implements JeuService {

            private final JeuRepository jeuRepository;
            private final EditeurRepository editeurRepository;

            public JeuServiceImpl(JeuRepository jeuRepository,
EditeurRepository editeurRepository) {
            this.jeuRepository = jeuRepository;
            this.editeurRepository = editeurRepository;
            }

            @Override
            public Jeu enregistrerJeu(Jeu jeu) {
            if (jeuRepository.findByNom(question.getNom())!=null) {
                  throw new JeuExistantException();
            }
            return jeuRepository.save(jeu);
```

```
            }

            @Override
            public Jeu recupererJeu(Long id) {
                  return
jeuRepository.findById(id).orElseThrow(JeuInexistantException::new);
            }

            @Override
            @Transactional(readOnly=true)
            public List<Jeu> recupereJeux() {
                  return jeuRepository.findAll();
            }

          }
```

La méthode d'enregistrement de ce service lève une exception maison (ce qui est bien un des rôles d'un service). L'exception maison existe dans le package exception :

```
package fr.clelia.avis.exception;

public class JeuExistantException extends RuntimeException {

      private static final long serialVersionUID = 1L;

}
```

10) Écrire le ou les contrôleurs Spring

Annoter chaque classe contrôleur avec @Controller.

10.1) (manière dépréciée) Demander à Spring l'injection des objets de type Service dans les contrôleurs grâce à l'annotation @Autowired placée sur chaque attribut.
NB : Chaque objet de type Service doit être annoté @Autowired.

Exemple:

```
          @Controller
          public class JeuController {

            @Autowired
            private JeuService jeuService;

            @Autowired
            private EditeurService editeurService;
```

10.2) (manière moderne, à préférer) Ajouter un constructeur dans le contrôleur avec en paramètre tous les objets que Spring doit injecter dans le contrôleur.

Exemple:

```
@Controller
public class JeuController {

   private final JeuService jeuService;
   private final EditeurService editeurService;

   public JeuController(JeuService jeuService, EditeurService
editeurService) {
         this.jeuService = jeuService;
         this.editeurService = editeurService;
   }

}
```

10.3) (manière encore plus moderne, à préférer) Ajouter l'annotation @AllArgsConstructor de Lombok qui va ajouter à la volée un constructeur avec en paramètre tous les objets que Spring doit injecter dans le contrôleur.

Exemple:

```
@Controller
@AllArgsConstructor
public class JeuController {

   private final jeuService jeuService;
   private final editeurService editeurService;

}
```

10.4) Ajouter les méthodes nécessaires pour traiter toutes les requêtes HTTP. Chacune de ces méthodes peut renvoyer un objet de type ModelAndView :

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/ModelAndView.html

```
@Controller
@AllArgsConstructor
public class JeuController {

   private final JeuService jeuService;
   private final EditeurService editeurService;

   @RequestMapping(value = { "/index", "/" })
   public ModelAndView accueil(@PageableDefault(size=5,
sort="nom") Pageable pageable) {
         // l'objet ModelAndView utilise la vue Thymeleaf index
         ModelAndView mav = new ModelAndView("index");
         // On place dans le model une page de jeux
         mav.addObject("pageDeJeux",
         jeuService.recupererJeux(pageable));
         return mav;
   }
}
```

11) Développer les vues Thymeleaf

Les vues doivent être ajoutées dans le dossier src/main/resources/templates.

Voici la vue index.hml :

```
<!DOCTYPE html>
<html lang="fr" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Accueil</title>
        <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.c
ss" rel="stylesheet" integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH"
crossorigin="anonymous">
</head>
<body>
<h1>Projet avis</h1>
<span th:text="${param.notification}"></span>
<form th:action="@{/connexion}" method="post">
    <label for="pseudo">Pseudo</label><input type="text" name="PSEUDO"
id="pseudo"><br>
    <label for="mot_de_passe">Mot de passe</label><input type="password"
name="MOT_DE_PASSE" id="mot_de_passe"><br>
    <input type="submit" value="Se connecter">
</form>
<a href="inscription">S'inscrire</a>
<br><br>
<h2>Liste des jeux</h2>
<table class="table table-striped table-hover">
<thead>
<tr>
<th>Nom <a th:href="@{'index?page=0&sort=nom'}">Trier</a></th>
<th>Editeur <a th:href="@{'index?page=0&sort=editeur.nom'}">Trier</a></th>
<th>Date de sortie <a th:href="@{'index?
page=0&sort=dateDeSortie,DESC'}">Trier</a></th>
</tr>
</thead>
<tr th:each="jeu:${pageDeJeux.getContent()}">
<td><span th:text="${jeu.nom}"></span></td>
<td><span th:text="${jeu.editeur.nom}"></span></td>
<td><span th:text="${jeu.dateDeSortie}"></span></td>
</tr>
</table>
<p>Page <span th:text="${pageDeJeux.number+1}"></span> sur <span th:text="$
{pageDeJeux.totalPages}"></span></p>
</body>
</html>
```

12) Mettre en œuvre une API REST avec des RestControllers

12.1) Annoter chaque classe contrôleur avec l'annotation @RestController.

« REST est un ensemble de contraintes architecturales. Il ne s'agit ni d'un protocole, ni d'une norme. Les développeurs d'API peuvent mettre en œuvre REST de nombreuses manières. Lorsqu'un client émet une requête par le biais d'une API RESTful, celle-ci transfère une représentation de l'état de la ressource au demandeur ou point de terminaison. Cette information, ou représentation, est fournie via le protocole HTTP dans l'un des formats suivants : JSON (JavaScript Object Notation), HTML, XLT, Python, PHP ou texte brut. Le langage de programmation le plus communément utilisé est JSON, car, contrairement à ce que son nom indique, il ne dépend pas d'un langage et peut être lu aussi bien par les humains que par les machines. »

Source : https://www.redhat.com/fr/topics/api/what-is-a-rest-api

De manière conventionnelle, vers une API REST :

- l'envoi d'une requête HTTP avec la méthode GET demande la récupération d'une ou plusieurs données
- l'envoi d'une requête HTTP avec la méthode POST demande l'ajout d'une donnée
- l'envoi d'une requête HTTP avec la méthode PUT demande la mise à jour complète d'une donnée
- l'envoi d'une requête HTTP avec la méthode PATCH demande la mise à jour partielle d'une donnée
- l'envoi d'une requête HTTP avec la méthode DELETE demande la suppression d'une donnée

```
@RestController
@RequestMapping("/api/jeux")
@AllArgsConstructor
@Validated
public class JeuRestController {

	private final JeuService jeuService;

	/**
	 * Cette méthode renvoie une page de jeux
	 *
	 * @param pageable correspond à une demande de page
	 * @param filtre correspond au filtre demandé
	 *
	 * @return une page de jeux
	 */
	@GetMapping("")
	public Page<Jeu> getJeux(
			@PageableDefault(size=15, page=0, sort="nom") Pageable
pageable,
			@RequestParam(required=false, defaultValue="") String
filtre
			) {
		return jeuService.recupererJeux(pageable, filtre);
	}
}
```

L'importation de pageable à choisir est celui de Spring Data et non celui de AWT qui est un vieux toolkit graphique.

12.2) Fournir les DTO

Pour des raisons de sécurité (voir la règle de SonarLint : https://sonarsource.atlassian.net/browse/RSPEC-4684) et pour éviter des récursivités dans le processus de sérialisation de Jackson, le développement des contrôleurs REST nécessite l'écriture de DTO.
Des mappers vont assurer la traduction d'un objet DTO en objet métier et inversement. Ces mappers sont regroupés dans un package mapper.

MapStruct (https://mapstruct.org/) est capable de générer les implémentations des mappers. Pour intégrer MapStruct, ajouter la dépendance ci-dessous dans la balise dependencies :

```
<dependency>
	<groupId>org.mapstruct</groupId>
	<artifactId>mapstruct</artifactId>
	<version>1.6.3</version>
</dependency>
```

En reconfigurant le plugin de compilation de Maven, les implémentations de mappers seront générées pendant la phase generate-sources de Maven dans le dossier target/generated-sources/annotations :

```
<build>
	<plugins>
		<plugin>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-maven-plugin</artifactId>
			<configuration>
				<excludes>
					<exclude>
						<groupId>org.projectlombok</groupId>
						<artifactId>lombok</artifactId>
					</exclude>
				</excludes>
			</configuration>
		</plugin>
		<plugin>
			<groupId>org.apache.maven.plugins</groupId>
			<artifactId>maven-compiler-plugin</artifactId>
			<configuration>
				<source>${java.version}</source>
				<target>${java.version}</target>
				<annotationProcessorPaths>
					<path>
						<groupId>org.projectlombok</groupId>
						<artifactId>lombok</artifactId>
						<version>${lombok.version}</version>
					</path>
					<path>
						<groupId>org.projectlombok</groupId>

<artifactId>lombok-mapstruct-binding</artifactId>
						<version>0.2.0</version>
					</path>
					<path>
						<groupId>org.mapstruct</groupId>
```

```
<artifactId>mapstruct-processor</artifactId>
                                                    <version>1.6.3</version>
                                            </path>
                                    </annotationProcessorPaths>
                            </configuration>
                    </plugin>
            </plugins>
</build>
```

12.3) Générer la documentation de l'API et la page Swagger

https://swagger.io/tools/swagger-ui/

Ajouter dans le fichier pom.xml, au niveau de la balise dependencies, la balise suivante :

```
<dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>2.8.6</version>
</dependency>
```

Cette dépendance fournit les annotations @Operation,@Parameter, etc de Swagger

Redémarrer l'application Spring Boot manuellement. La documentation Swagger est désormais accessible à partir de l'URL suivante :
http://localhost:8080/swagger-ui/index.html

Cette page présente les opérations de l'API de manière très lisible :

La documentation OpenAPI (au format JSON) est disponible à l'adresse suivante :
http://localhost:8080/v3/api-docs

Par défaut, chaque opération requiert un clic sur le bouton « Try it out ». Pour se dispenser de cliquer sur ce bouton, la ligne ci-dessous est requise dans le fichier src/main/resources/application.properties :

```
springdoc.swagger-ui.try-it-out-enabled=true
```

13) Écriture des tests

13.1) Ajouter un fichier application.properties dans un nouveau dossier src/test/resources

Ce fichier peut contenir des propriétés qui configurent une base H2 en mémoire dont la durée de vie sera limitée à la phase de tests :

```
spring.datasource.url=jdbc:h2:mem:avis_test
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.show-sql=true
```

Cette configuration servira exclusivement pendant la phase de tests. Sans la présence du fichier src/test/resources/application.properties, Maven utilisera le fichier src/main/resources/application.properties pour lancer les tests.

13.2) Écrire les classes de tests dans le dossier src/test/java

Le package repository accueillera les tests sur les interfaces repository. Ces classes de test pourront être annotées @SpringBootTest ou @DataJpaTest.

Le package service accueillera les tests sur les classes de la couche service. Ces classes de test seront si besoin annotées @SpringBootTest.

Le package controller accueillera les tests sur les classes de la couche controller. Ces classes de test seront annotées @SpringBootTest ou @WebMvcTest.

14) Générer la Javadoc

Il est possible de confier la génération de la Javadoc et des informations sur le projet Maven en ajoutant dans le fichier pom.xml la balise reporting :

```
	<reporting>
		<plugins>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-project-info-reports-
plugin</artifactId>
				<version>3.8.0</version>
			</plugin>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-javadoc-plugin</artifactId>
				<version>3.10.1</version>
			</plugin>
		</plugins>
		<outputDirectory>doc</outputDirectory>
	</reporting>
```

Puis lancer maven site.

15) Lancer l’application

L’application se lance à partir du goal Maven : ./mvnw spring-boot:run

Annexes :

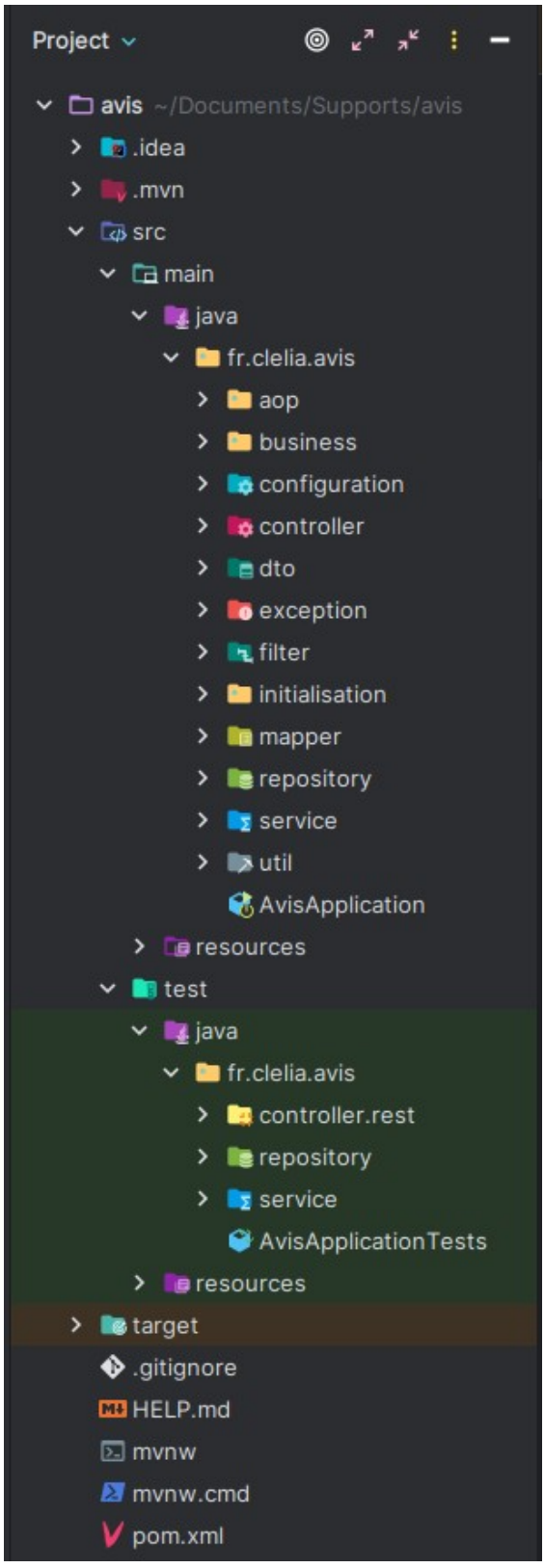
A) Pour obtenir un script de création des tables en base, les deux lignes suivantes doivent être ajoutées dans le fichier src/main/resources/application.properties :

spring.jpa.properties.javax.persistence.schema-generation.scripts.action=create
spring.jpa.properties.javax.persistence.schema-generation.scripts.create-target=src/main/resources/create.sql

Les valeurs possibles d’action sont : create, update, drop

A noter : si ces deux lignes sont présentes dans le fichier de configuration, les ordres de création de tables ne seront plus envoyés à la base par JPA.

B) Capture d’écran présentant l’arborescence du projet avis dans IntelliJ 2024.3.5 (Ultimate Edition) :



C) Pour modifier la stratégie de nommage des tables et des colonnes en base, il suffit d'ajouter la ligne suivante :

spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

Avec cette stratégie de nommage :
- les noms de tables auront une majuscule à chaque mot : exemple : TypeClient
- le nom des colonnes sera identique au nom des attributs de la classe : dateHeureCreation

D) Pour autoriser le téléversement de fichiers (dont la taille est supérieure à 1 Mo) sur le serveur, ces deux lignes sont indispensables :

spring.servlet.multipart.max-file-size=8MB
spring.servlet.multipart.max-request-size=10MB

Les lignes ci-dessus ont le même effet que cette classe de configuration :

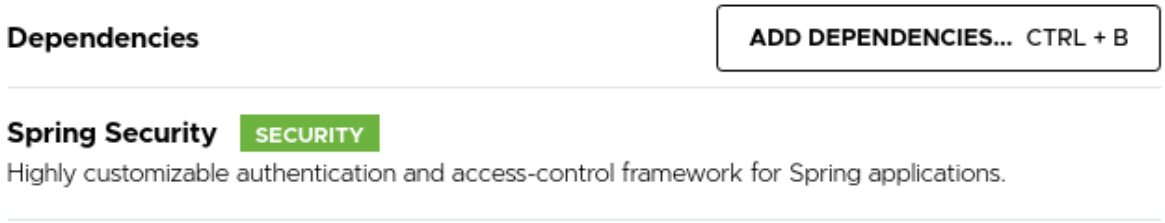
```
package fr.clelia.avis.configuration;

import jakarta.servlet.MultipartConfigElement;

import org.springframework.boot.web.servlet.MultipartConfigFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.util.unit.DataSize;

@Configuration
public class TeleversementConfiguration {

	@Bean(name="uploadConfig")
	MultipartConfigElement multipartConfigElement() {
		MultipartConfigFactory factory = new MultipartConfigFactory();
		factory.setMaxFileSize(DataSize.ofMegabytes(8));
		factory.setMaxRequestSize(DataSize.ofMegabytes(10));
		return factory.createMultipartConfig();
	}

}
```

La configuration via la classe de Java l'emporte sur la configuration des fichiers properties et YAML.

E) Pour modifier le nombre de connexions créées entre l'application et la base (via le connection pool Hikari présent par défaut dans l'application Spring Boot), il faut écrire :

```
spring.datasource.hikari.maximum-pool-size=15
```

F) Pour intégrer Spring Security au projet, ajouter la dépendance associée :



Le fichier pom.xml contiendra la dépendance ci-dessous :

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

Dans la classe exécutable (la classe qui contient la méthode main()) ajouter un bean chargé de chiffrer les mots de passe avec Bcrypt :

```
@Bean
public PasswordEncoder encoder() {
	return new BCryptPasswordEncoder();
}
```

Un des services doit implémenter l'interface UserDetailsService :
https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetailsService.html

Jusqu'à la version 5.6 de Spring Security, il fallait ajouter une classe de configuration héritant de WebSecurityConfigurerAdapter.

Depuis la version 5.7 de Spring Security, il n'est plus recommandé de créer une classe de configuration qui hérite de WebSecurityConfigurerAdapter :
https://spring.io/blog/2022/02/21/spring-security-without-the-websecurityconfigureradapter

```
@Configuration
@AllArgsConstructor
public class SecurityConfiguration {

	private UserDetailsService userDetailsService;
	private PasswordEncoder passwordEncoder;

@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

	 http.csrf(csrf -> csrf.ignoringRequestMatchers("/h2-console/**"))
		.authorizeHttpRequests((authorize) -> authorize
			.requestMatchers("/swagger-ui/**").permitAll()
			.requestMatchers("/index").permitAll()
			.requestMatchers("/api/editeurs/**").hasRole("ADMIN")
			.requestMatchers("/h2-console/**").authenticated()
			.requestMatchers(HttpMethod.POST,
			    "/jeux").hasRole("ADMIN")
			.anyRequest().authenticated()
		)
		.httpBasic(withDefaults())
		.formLogin(withDefaults())
		.headers(headers -> headers.frameOptions(frameOptions ->
			frameOptions.disable()));

    return http.build();
```

}

Le formulaire de connexion doit impérativement contenir un champ de saisie dont le nom est username et un champ de saisie dont le nom est password :

```
<form action="login" method="post">
      <input type="email" name="username" placeHolder="Email" required><br>
      <input type="password" name="password" placeHolder="Mot de Passe"
required><br>
      <input type="submit" value="Connexion">
</form>
```

G) Pour que Spring lance des tâches programmées, la classe contenant la méthode main doit être annotée @EnableScheduling. Chaque méthode que Spring doit invoquer automatiquement doit être annotée @Scheduled.

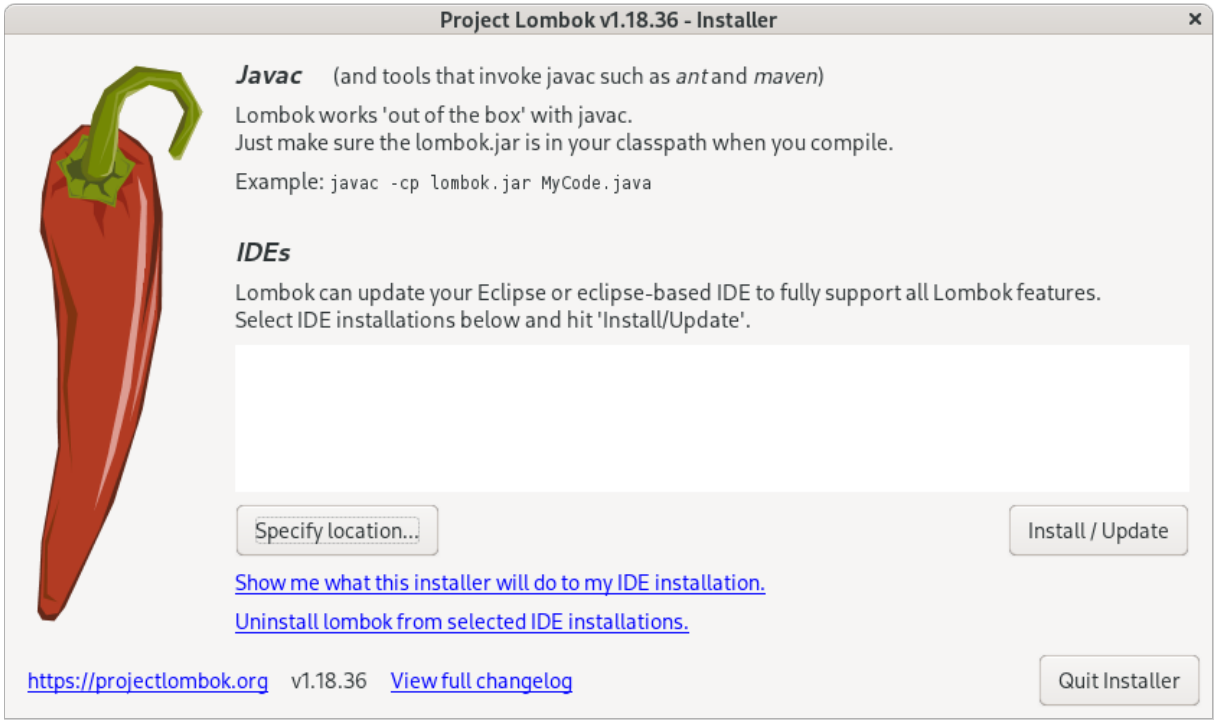
H) Pour modifier le port sur lequel le serveur Tomcat écoute :

server.port=8180

I) Pour intégrer Lombok dans Eclipse, aller sur le menu Help / Install New Software.
Cliquer sur Add, dans le champ Location, indiquer : https://projectlombok.org/p2
Cliquer sur Add puis selectionner Lombok 1.18.36.
Source : https://projectlombok.org/setup/eclipse

Si cela ne marche pas :
Quitter Eclipse, lancer un terminal puis exécuter le jar de lombok disponible dans .m2/repository/org/projectlombok/lombok/1.18.36/ : java -jar lombok-1.18.36.jar

Si Lombok ne détecte pas l'IDE. Relancer l'IDE. Le chemin où Eclipse est installé est précisé sur l'onglet Configuration de la modale : Help / About Eclipse / Installation Details.



J) Définition de deux profiles dans le fichier src/main/resources/application.properties

```
spring.profiles.active=dev

#---
spring.config.activate.on-profile=dev
logging.level.root=INFO
logging.level.org.springframework=DEBUG
logging.file.name=log/avis_dev_log
logging.pattern.console= %d %p %c{1.} [%t] %m%n
#---
spring.config.activate.on-profile=prod
logging.level.root=WARN
logging.level.org.springframework=WARN
logging.file.name=log/avis_prod_log
logging.pattern.console= %d %p %c{1.} [%t] %m%n
```

K) Exemple de fichier application.properties utilisant une source de données MySQL :

```
server.port=8180

spring.datasource.url=jdbc:mysql://localhost:3306/avis?useSSL=false
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.generate_statistics=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.generate-ddl=true
```

```
spring.h2.console.enabled=true

spring.mvc.view.suffix=.jsp
spring.mvc.view.prefix=/WEB-INF/

logging.level.root=WARN
logging.level.org.springframework=WARN
logging.file.name=log/avis_log
logging.pattern.console= %d %p %c{1.} [%t] %m%n

spring.data.rest.detection-strategy=annotated
spring.data.rest.base-path=/api-autogeneree/

management.endpoint.info.enabled=true
management.endpoints.web.base-path=/
management.endpoints.web.exposure.include=beans

server.error.path=/erreur

spring.servlet.multipart.max-file-size=8MB
spring.servlet.multipart.max-request-size=10MB
```

L) Version de dépendances utilisées par Spring Boot 3.4.4 sortie le 20/03/2025 :
https://spring.io/blog/2025/03/20/spring-boot-3-4-4-available-now

Spring Framework : 6.2.5
AspectJ : 1.9.23
Hibernate : 6.6.11
Hibernate Validator : 8.0.2
Spring Security : 6.4.4
Log4J : 2.24.3
Tomcat : 10.1.39
Jackson : 2.18.3
Lombok : 1.18.36
H2 : 2.3.232
Mockito : 5.14.2
JUnit : 5.11.4
Swagger-ui : 5.13.0
Thymeleaf : 3.1.3

Source : https://github.com/spring-projects/spring-boot/releases/tag/v3.4.4