

Spring

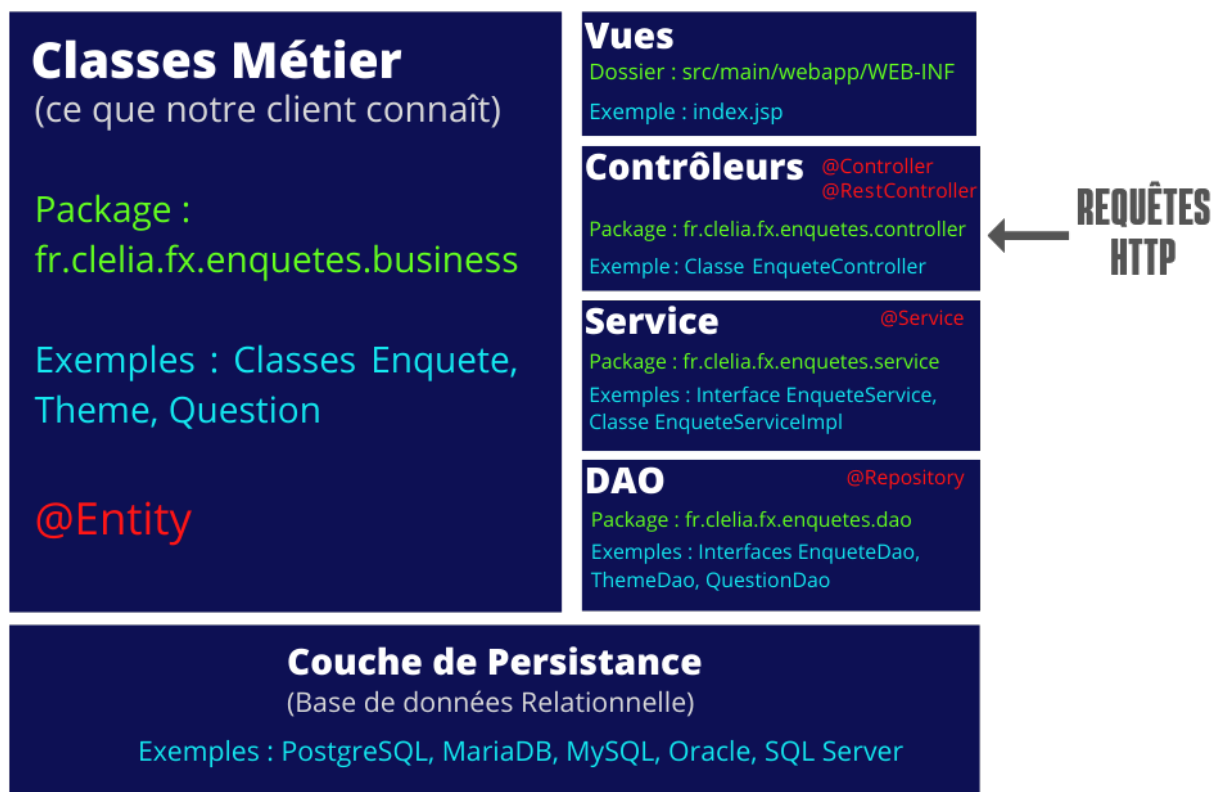
Cook Book

Auteur du document : François-Xavier COTE (fxcote@clelia.fr)

Version du document : 1.45 en date du 26/04/2023

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 enquetes
- utilisant des JSPs pour le front avec la bibliothèque JSTL : <https://jakarta.ee/specifications/tags/> et la bibliothèque de balises JSP de Spring
- 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 EnqueteController a besoin d'un service EnqueteService, il déclare cette dépendance en faisant référence à l'interface et non à l'implémentation du service (EnqueteServiceImpl) :

```
@Controller
public class EnqueteController {

    private final EnqueteService enqueteService;

}
```

1) Générer le projet Maven à l'aide de Spring Initializr : <https://start.spring.io>



Project

☐ Gradle - Groovy
 ☐ Gradle - Kotlin
 ☒ Maven

Language

☒ Java
 ☐ Kotlin
 ☐ Groovy

Spring Boot

☐ 3.1.0 (SNAPSHOT)
 ☐ 3.1.0 (RC1)
 ☐ 3.1.0 (M2)
 ☐ 3.0.7 (SNAPSHOT)
 ☐ 3.0.6
 ☐ 2.7.12 (SNAPSHOT)
 ☒ 2.7.11

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 20 ☒ 17 ☐ 11 ☐ 8

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Data JPA

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database

SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Validation

validation

Bean Validation with Hibernate validator.

Lombok

DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

Spring Boot Actuator

OPS

Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.

Rest Repositories

WEB

Exposing Spring Data repositories over REST via Spring Data REST.

Spring Boot DevTools

DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...

NB : Huit dépendances ont été ajoutées.

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. La version de chaque dépendance (du projet parent) est précisée sur cette page :

<https://docs.spring.io/spring-boot/docs/current/reference/html/dependency-versions.html>

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. 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".

Afin d'obtenir des classes persistantes, autrement dit annotées avec @Entity de JPA (Jakarta Persistence API : <https://jakarta.ee/specifications/persistence/2.2/>), 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) Décompresser le fichier zip dans le workspace d'Eclipse

3) Dans Eclipse, importer le projet Maven en utilisant l'assistant : File / Import / Existing Maven projects. Dans IntelliJ ouvrir le projet Maven. En cas d'erreur dans IntelliJ supprimer le dossier .idea et ouvrir de nouveau le projet.

4) Ajouter dans le fichier pom.xml, au niveau de la balise dependencies, les deux balises suivantes :

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
```

La première balise incorpore Jasper au projet. Jasper convertit les JSPs en servlets. La seconde balise intègre la bibliothèque balises JSTL. Ce faisant les JSPs auront accès à toutes les balises de cette bibliothèque (if, when, choose, forEach, etc).

5) 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
```

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

A noter : les autres valeurs de ddl-auto sont : `none`, `validate`, `create`, et `create-drop`.
Avec la dernière valeur, Hibernate essaie de supprimer toutes les tables avant de les créer ce qui se révèle très utile en phase de développement.

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

```
spring.datasource.url=jdbc:h2:mem:enquetes
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

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.

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

```
spring.datasource.url=jdbc:h2:~/enquetes
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

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



```
spring.datasource.url=jdbc:postgresql://localhost:5432/enquetes
spring.datasource.username=postgres
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQL10Dialect
```

5.4) Pour une base MySQL nommée enquetes, le fichier de configuration doit également inclure :



```
spring.datasource.url=jdbc:mysql://localhost:3306/enquetes?useSSL=false
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

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

```
management.endpoint.info.enabled=true
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>

5.6) Pour demander à Spring la gestion d'un fichier de journalisation :

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

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

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 :

```
2022-01-07 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
```

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>

6) Écrire/Générer les classes business:

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.fx.enquetes.business.Enquete

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 Question {

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

    private String libelle;

    @ManyToOne
    private Enquete enquete;

    public Question() {
    }
    ...
}
```

```
}
```

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
@Getter
@Setter
@EqualsAndHashCode
@ToString
public class Question {

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

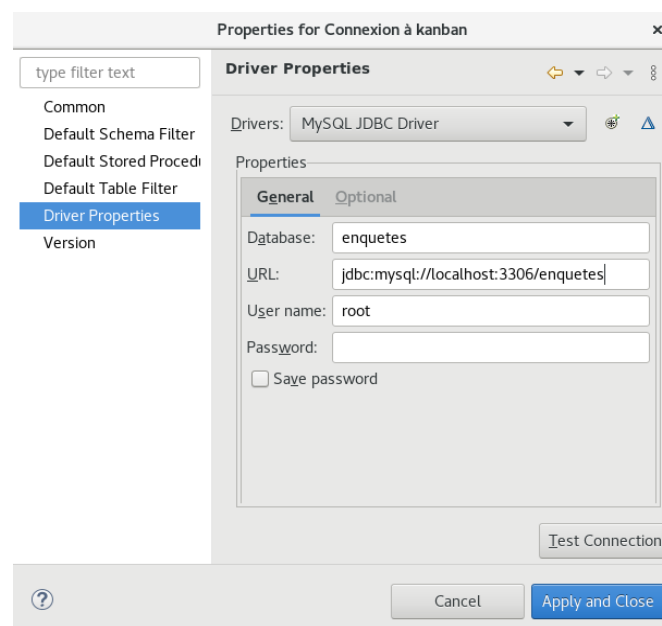
    private String libelle;

    @ManyToOne
    private Enquete enquete;
    ...
}
```

6.2) Soit en **Database First**:

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

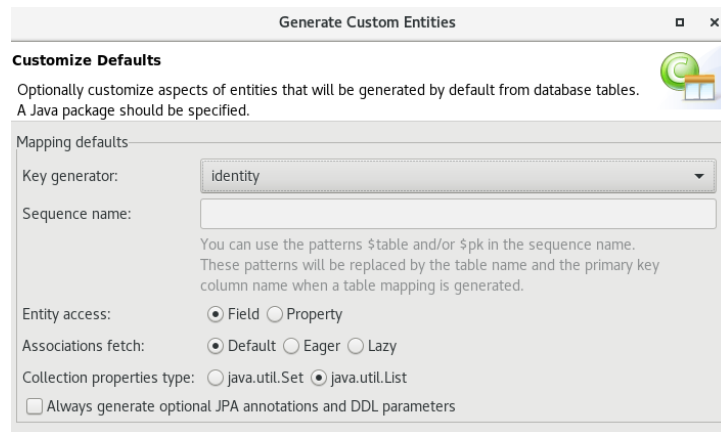
6.2.2) ajouter une connexion à la base de données



6.2.3) cliquer-droit sur le projet: configure / convert to JPA project

6.2.4) JPA est coché, cliquer sur Next

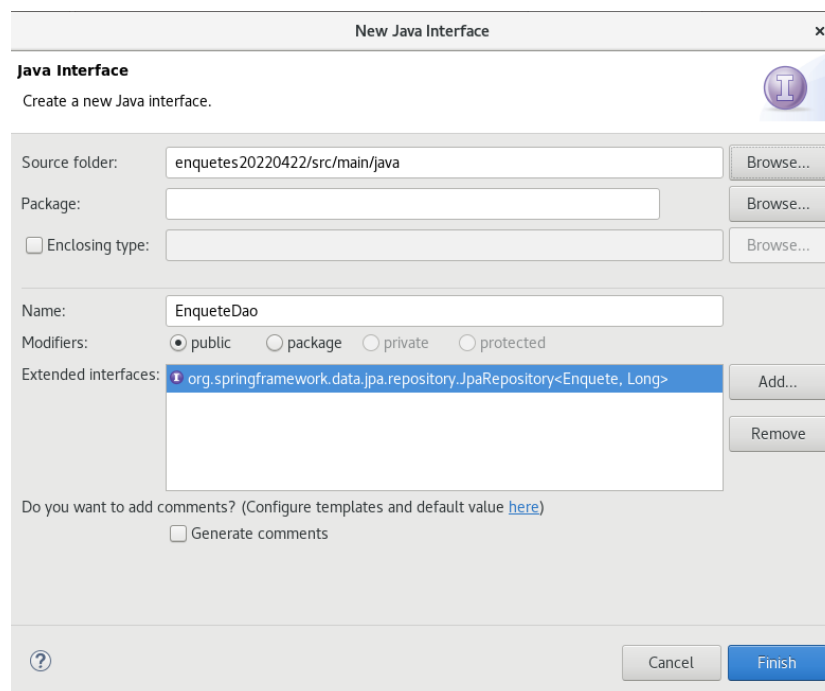
- 6.2.5) Choisir Generic 2.1 et disable user library puis cliquer sur Finish
- 6.2.6) Cliquer droit sur le projet JPA Tools / Generate Entities from Tables
- 6.2.7) Sélectionner toutes les tables, cliquer sur Next 2 fois
- 6.2.8) Sur la fenêtre « Customize Defaults » choisir identity comme Key generator 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 avec le reverse engineer de StarUML et placer le fichier .mdj ainsi qu'une version PNG du diagramme dans un dossier nommé doc

8) Écrire les interfaces DAO en sélectionnant dans le menu File / New / Interface. Chaque interface hérite de JpaRepository :



Exemple:

```
public interface EnqueteDao extends JpaRepository<Enquete, 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 DAO peuvent être générées en choisissant une ou plusieurs classes puis new Spring Data Repository.

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>

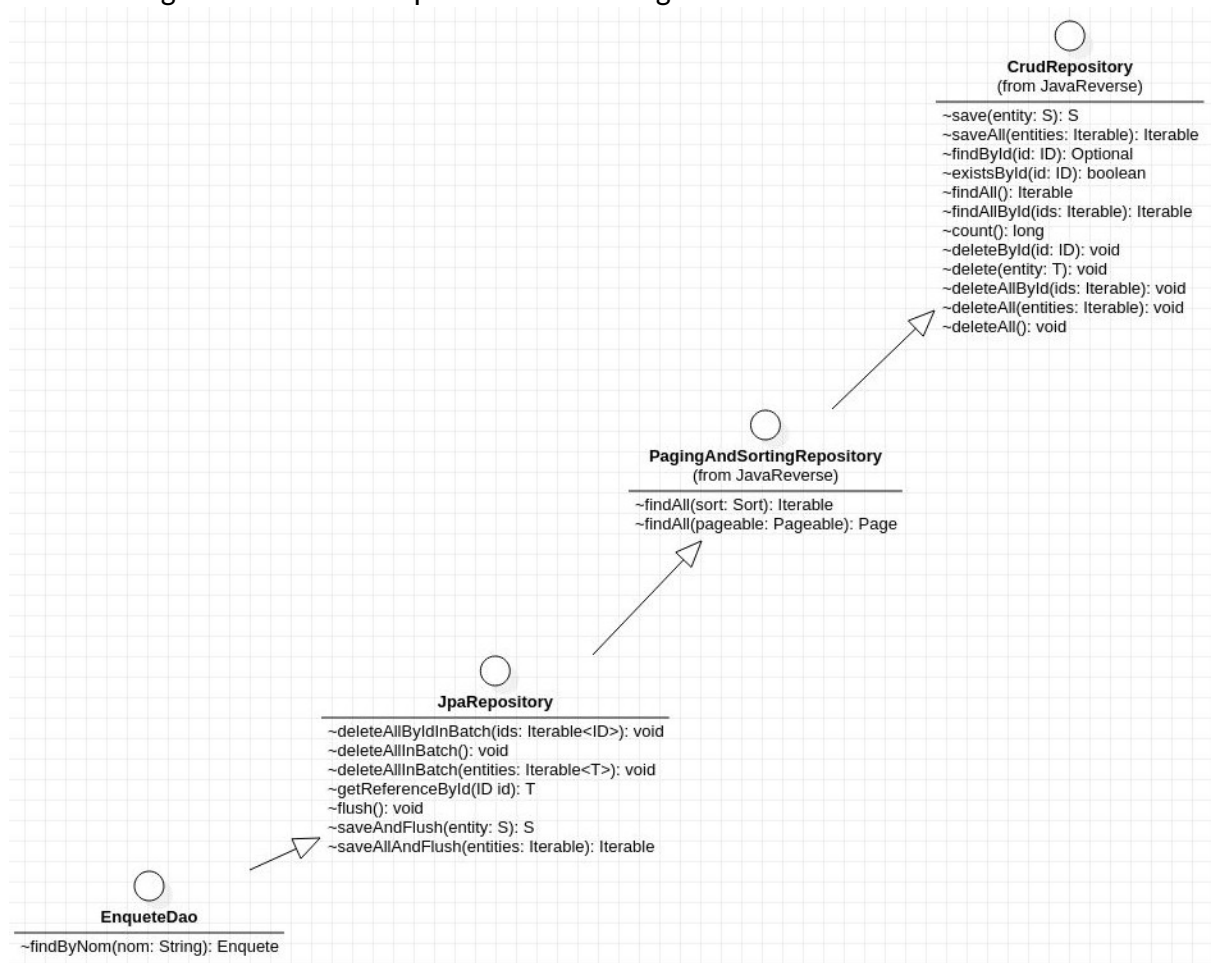
L'interface JpaRepository hérite de l'interface PagingAndSortingRepository :

<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html>

L'interface PagingAndSortingRepository hérite de l'interface CrudRepository :

<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 4 interfaces :



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

```
@Query("FROM Theme t ORDER BY size(t.enquetes) DESC")
List<Theme> findThemesSortedByNbEnquetes();
```

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 Enquete WHERE theme_id=:idTheme",
nativeQuery=true)
List<Enquete> findByIdTheme(@Param("idTheme") Long idTheme);
```

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

```
List<Enquete> findByTheme(Theme theme);
```

Les mots clés autorisés dans le nom des méthodes sont résumés sur la table suivante :

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

9) Écrire les interfaces puis les classes dans le paquetage service. Annoter chaque classe service avec le stéréotype Spring @Service et demander à Spring l'injection des DAO par l'écriture d'un constructeur ayant en paramètre les objets que Spring doit injecter dans le service.

Exemple:

```
@Service
public class QuestionServiceImpl implements QuestionService {

    private final QuestionDao questionDao;
    private final EnqueteDao enqueteDao;

    public QuestionServiceImpl(QuestionDao questionDao, EnqueteDao
        enqueteDao) {
        super();
        this.questionDao = questionDao;
        this.enqueteDao = enqueteDao;
    }

    @Override
    public Question enregistrerQuestion(Question question) {
        if (questionDao.findByLibelle(question.getLibelle())!=null) {
            throw new QuestionExistanteException();
        }
        questionDao.save(question);
        return question;
    }
}
```

```

@Override
public Question recupererQuestion(Long id) {
    return questionDao.findById(id).orElse(null);
}

@Override
@Transactional(readOnly=true)
public List<Question> recupereQuestions() {
    return questionDao.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.fx.enquetes.exception;

public class QuestionException 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 EnqueteController {

    @Autowired
    private EnqueteService enqueteService;

    @Autowired
    private QuestionService questionService;
}

```

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 EnqueteController {

    private final EnqueteService enqueteService;
    private final QuestionService questionService;
}

```

```

        public EnqueteController(EnqueteService enqueteService,
QuestionService questionService) {
            super();
            this.enqueteService = enqueteService;
            this.questionService = questionService;
        }
    }
}

```

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 EnqueteController {

    private final EnqueteService enqueteService;
    private final QuestionService questionService;

}

```

10.4) Ajouter les méthodes nécessaires pour traiter toutes les requêtes HTTP. Chacune de ces méthodes doit 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 EnqueteController {

    private final EnqueteService enqueteService;
    private final QuestionService questionService;

    @RequestMapping(value = { "/index", "/" })
    public ModelAndView accueil() {
        ModelAndView mav = new ModelAndView("index");
        mav.addObject("enquetes",
enqueteService.recupererEnquetes());
        return mav;
    }

}

```

11) Ajouter un dossier nommé `src/main/webapp/WEB-INF`

12) Écrire les JSPs dans le dossier `src/main/webapp/WEB-INF` en cliquant-droit sur ce dossier, New / JSP File. En plaçant les JSPs dans ce dossier, elles ne sont pas accessibles publiquement, seuls les contrôleurs peuvent les utiliser.

Pour bénéficier des balises JSTL (Java Standard Tag Library) ajouter la directive suivante dans la JSP :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

Grâce à cette bibliothèque de balises, l'écriture de la vue est simplifiée :

- `<c:if>` pour ajouter un branchement.

Exemple : on teste si l'attribut `erreur` n'est pas nul. Dans l'affirmative on affiche cet attribut dans une balise `h2` :

```
<c:if test="${erreur ne null}"><h2>${erreur}</h2></c:if>
```

NB : le mot clé **ne** signifie « not equals ». Il rend la condition du test plus lisible. Il existe également : **eq**, **gt**, **lt**, **ge** et **le**.

- `<c:forEach>` pour parcourir une collection. Cette balise doit obligatoirement avoir deux attributs `items` et `var`. L'attribut `items` correspond à la collection à parcourir. L'attribut `var` correspond au nom de la variable de boucle i.e. la variable locale à la boucle.

Exemple 1 : on parcourt une liste d'objets de type Utilisateur nommée `utilisateurs`, la variable de boucle se nomme `utilisateur`.

```
<ul>
<c:forEach items="${utilisateurs}" var="utilisateur">
  <li>${utilisateur.prenom}</li>
</c:forEach>
</ul>
```

Exemple 2 : on parcourt une liste d'objet de type Ville nommée `villes`, la variable de boucle se nomme `ville`.

```
<select name="ID_VILLE">
<c:forEach items="${villes}" var="ville">
  <option value="${ville.id}">${ville.nom}</option>
</c:forEach>
</select>
```

Exemple 3 : on parcourt une liste d'objet de type Ville nommée `villes`, la variable de boucle se nomme `ville`. La balise `forEach` déclare une variable `compteur` qui s'auto-incrémente à chaque itération :

```
<select name="ID_VILLE">
<c:forEach items="${villes}" var="ville" varStatus="compteur">
  <option value="${ville.id}">${compteur.index} : $
{ville.nom}</option>
</c:forEach>
</select>
```

Exemple 4 : on fait évoluer un compteur allant de 1 à nbSemaines. A chaque itération, le compteur est incrémenté :

```
<c:forEach var="i" begin="1" end="{nbSemaines}" step="1">
    ${i}
</c:forEach>
```

- <c:choose>, <c:when> et <c:otherwise> pour les branchements complexes

Exemple :

```
<c:choose>
<c:when test="{article.stock<5 && article.stock>0}">
stock faible</c:when>
<c:when test="{article.stock==0}">
stock épuisé</c:when>
<c:otherwise>stock disponible</c:otherwise>
</c:choose>
```

- <c:set> pour définir des variables locales

Exemple :

```
<c:set var="emoji" value="&#x1F7E0;" />
```

Pour bénéficier des balises de la bibliothèque JSP de Spring, ajouter la directive suivante dans la JSP :

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
```

Grâce à cette bibliothèque de balises, l'écriture du formulaire HTML est simplifiée :

- <form:form> pour ajouter un formulaire HTML dans lequel un objet métier sera injecté. Cet objet métier est désigné « model attribute ».

Exemple : on ajoute un formulaire qui reçoit un objet métier utilisateur. Les données seront envoyées dans le corps de la requête HTTP car l'attribut method a la valeur post. En cliquant sur le bouton d'inscription, les données seront envoyées à l'URL inscription

```
<form:form modelAttribute="utilisateur" action="inscription"
method="post">
```

- <form:label> pour ajouter une étiquette

Exemple : on ajoute une étiquette associé à l'adresse email de l'utilisateur :

```
<form:label path="email">Email</form:label>
```

- <form:input> pour ajouter un champ de saisie

Exemple : on ajoute un champ de saisie qui correspond à l'adresse email de l'utilisateur :

```
<form:input path="email" />
```

- <form:password> pour ajouter un champ de saisie de type password

Exemple : on ajoute un champ de saisie qui correspond au mot de passe de l'utilisateur :

```
<form:password path="motDePasse" />
```

- <form:errors> pour ajouter un espace dans lequel Spring pourra afficher les erreurs de validation obtenues en essayant de valider les données saisies sur le formulaire avec les contraintes de validation exprimées dans la classe métier Utilisateur

Exemple :

```
<form:errors path="email" cssClass="erreur" />
```

- <form:select> pour ajouter une liste déroulante

Exemple :

```
<form:select path="ville">
```

- <form:option> pour ajouter une option à une liste déroulante

Exemple :

```
<form:option value="">Merci de choisir une ville</form:option>
```

- <form:options> pour ajouter une collection d'options à une liste déroulante

Exemple : on demande à Spring de créer une balise option pour chaque objet de la liste villes. La valeur de l'option contiendra l'id de la ville. L'option affichera le nom de la ville :

```
<form:options items="${villes}" itemValue="id"
itemLabel="nom"></form:options>
```

- <form:button> pour ajouter un bouton au formulaire

Exemple :

```
<form:button>Enregistrer</form:button>
```

13) Pour mettre en œuvre une API REST, écrire le ou les contrôleurs Spring. 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, XML, 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/")
@AllArgsConstructor
@Validated
public class UtilisateurRestController {

    private final UtilisateurService utilisateurService;
    private final ThemeService themeService;

    /**
     * Cette méthode renvoie une page d'utilisateurs
     *
     * @param pageable correspond à une demande de page
     * @return une page d'utilisateurs
     */
    @GetMapping("utilisateurs")
    public Page<Utilisateur> utilisateursPages(
        @PageableDefault(size=15, page=0, sort="email") Pageable
        pageable,
        @RequestParam(required=false, defaultValue="") String
        filtre
    ) {
        return utilisateurService.recupererUtilisateurs(pageable,
        filtre);
    }
}
```


NB : l'importation de pageable à choisir est celui de Spring et non celui de AWT qui est un vieux toolkit graphique.

14) 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 :

```
spring.datasource.url=jdbc:h2:mem:enquetes_test
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

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.

15) Écrire les tests dans le dossier src/test/java

Le package dao accueillera les tests sur les interfaces DAO. 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 annotées @SpringBootTest.

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

16) Générer la documentation de l'API et la page Swagger-UI

<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-ui</artifactId>
  <version>1.7.0</version>
</dependency>
```

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 ressources de l'API de manière très lisible :

PUT	/api/villes/{id}/{nom}	▼
GET	/api/villes	▼
POST	/api/villes	▼
POST	/api/villes/{nom}	▼
GET	/api/villes/{id}	▼
DELETE	/api/villes/{id}	▼

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

17) 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>2.6</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>3.3.2</version>
    </plugin>
  </plugins>
  <outputDirectory>doc</outputDirectory>
</reporting>
```

Puis lancer maven site.

18) Lancer l'application avec le goal Maven : ./mvnw spring-boot:run

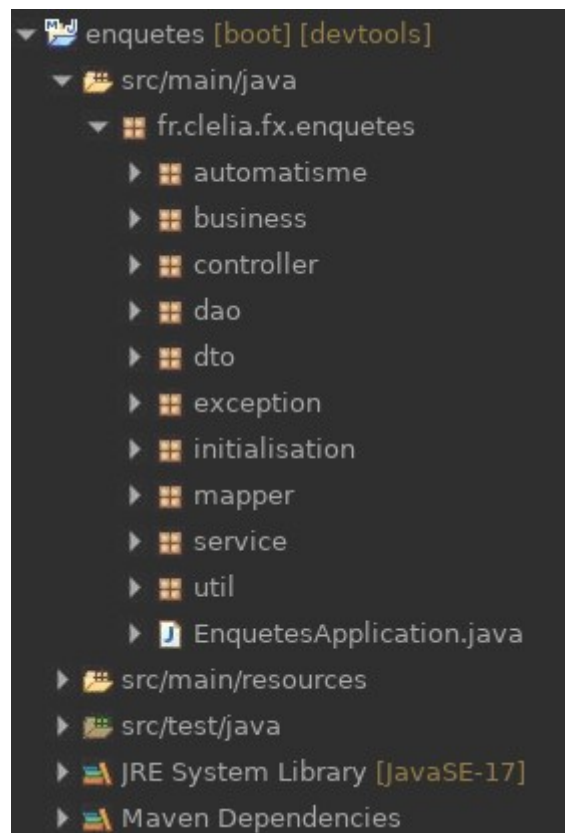
Annexes :

A) Pour obtenir un script de création et ou de suppression 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/script.sql
```

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 enquetes dans l'IDE Eclipse 2023-03:



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
```

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 :

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Security

SECURITY

Highly customizable authentication and access-control framework for Spring applications.

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](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().disable()

        .authenticationManager(new
CustomAuthenticationManager(userDetailsService, passwordEncoder))

        .formLogin()
            // On fait référence à une URL
            .loginPage("/index")
            .loginProcessingUrl("/login")
            .defaultSuccessUrl("/enquetes")
            .failureForwardUrl("/index?notification=Email%20ou%20mot%20de
%20passe%20incorrect")
            .and()
            .logout()
            .logoutUrl("/deconnexion")
            .logoutSuccessUrl("/index?notification=Au%20revoir")
            .and()
            .authorizeRequests()
            .antMatchers("/h2-console").permitAll()
            .antMatchers("/enquetes").authenticated()
            .antMatchers("/enquete").authenticated()
            .antMatchers("/questions").authenticated()
            .antMatchers("/question").authenticated()
            // Pour la console H2 (à ne pas utiliser en prod)
            .and()
            .headers().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>

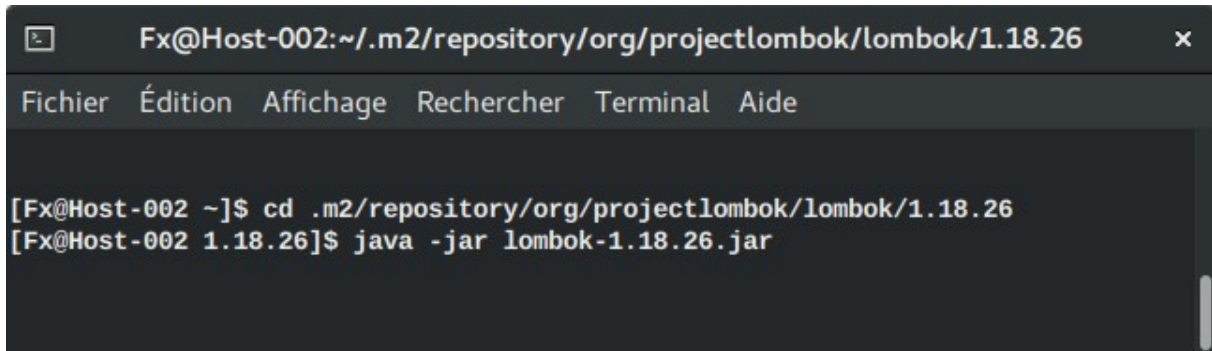
```

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=8280
```

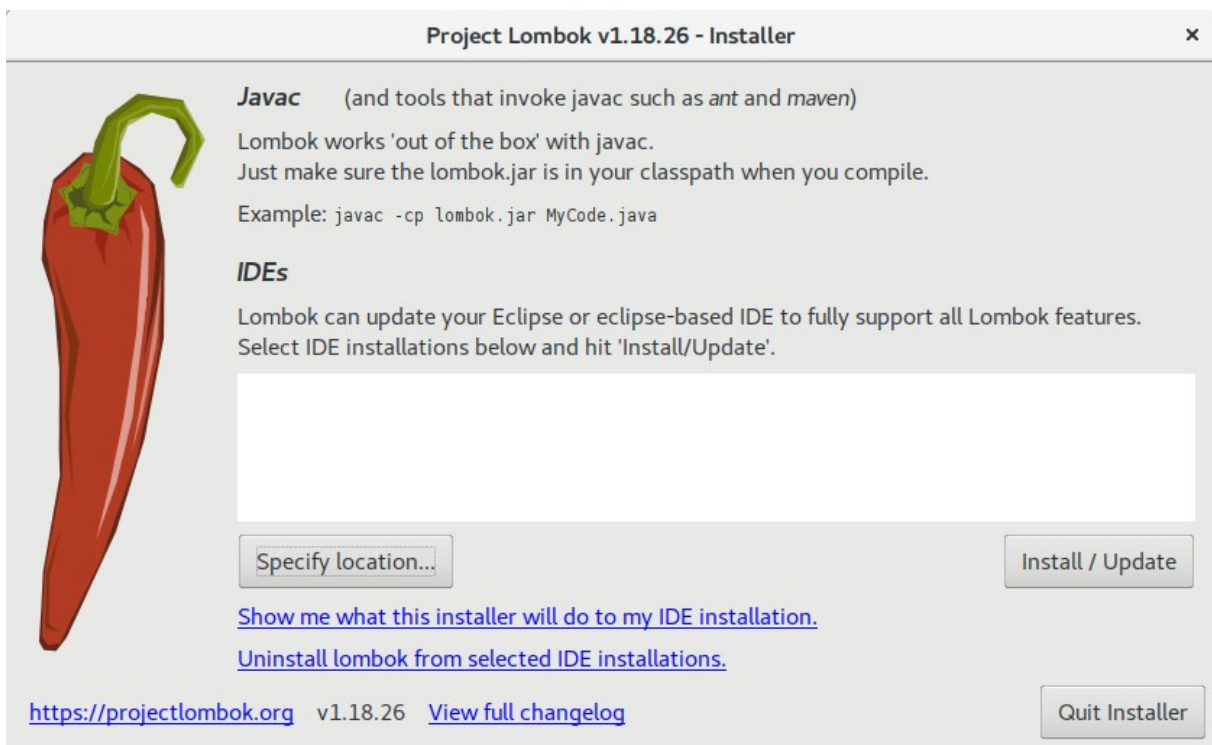
I) Pour intégrer Lombok dans l'IDE. Quitter l'IDE puis exécuter le jar de lombok disponible dans `.m2/repository/org/projectlombok/lombok/1.18.26/` : `java -jar lombok-1.18.26.jar`



```

Fx@Host-002:~/m2/repository/org/projectlombok/lombok/1.18.26
Fichier  Édition  Affichage  Rechercher  Terminal  Aide

[Fx@Host-002 ~]$ cd .m2/repository/org/projectlombok/lombok/1.18.26
[Fx@Host-002 1.18.26]$ java -jar lombok-1.18.26.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) Exemple de fichier `application.properties` utilisant une source de données MySQL :

`server.port=8280`

`spring.datasource.url=jdbc:mysql://localhost:3306/enquetes?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.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect  
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/enquetes_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
```

K) Version de dépendances utilisées par Spring Boot 2.7.11 sortie le 20/04/2023 :

<https://spring.io/blog/2023/04/20/spring-boot-2-7-11-available-now-fixing-cve-2023-20873>

```
Spring Framework : 5.3.27  
Hibernate : 5.6.15  
Hibernate Validator : 6.2.5  
Spring Security : 5.7.8  
Log4J : 2.17.2  
Tomcat : 9.0.74  
Jackson : 2.13.5  
Lombok : 1.18.26  
H2 : 2.1.214  
Mockito : 4.5.1  
JUnit : 5.8.2
```

Source : <https://github.com/spring-projects/spring-boot/releases/tag/v2.7.11>