Université Sidi Mohamed Ben Abdellah
Faculté des Sciences Dhar El Mahraz
Département : Informatique
Filière : Qualité du Logiciel (MQL)

De Java EE à Spring Boot : Évolution des architectures et des pratiques

Réalisé par :

BENZARIYA Saida

Proposé par :

M.Noureddine Chenfour

Année Universitaire : 2024/2025

Introduction générale

Le développement d'applications Java d'entreprise a évolué significativement au fil des années, passant de l'approche Java EE native, souvent lourde et complexe, à des frameworks plus modernes comme Spring et Spring Boot. Ce rapport vise à explorer ces différentes approches en mettant en lumière leur architecture, leur configuration, ainsi que les outils et technologies qu'elles intègrent. L'objectif est de comparer la flexibilité, la facilité de développement, la maintenabilité et l'expérience développeur qu'offrent ces solutions afin d'identifier les avantages et inconvénients de chacune.

Table des matières

1 Version JEE Native : Architecture MVC classique

Le projet `p02-jee-mvc2` suit une architecture en couches (pattern MVC), typique des applications Java EE classiques. Cette architecture permet une séparation claire des responsabilités et facilite la maintenabilité.

1.1 Architecture Générale :

Ton application suit une architecture en couches (très proche de Spring MVC) composée de :

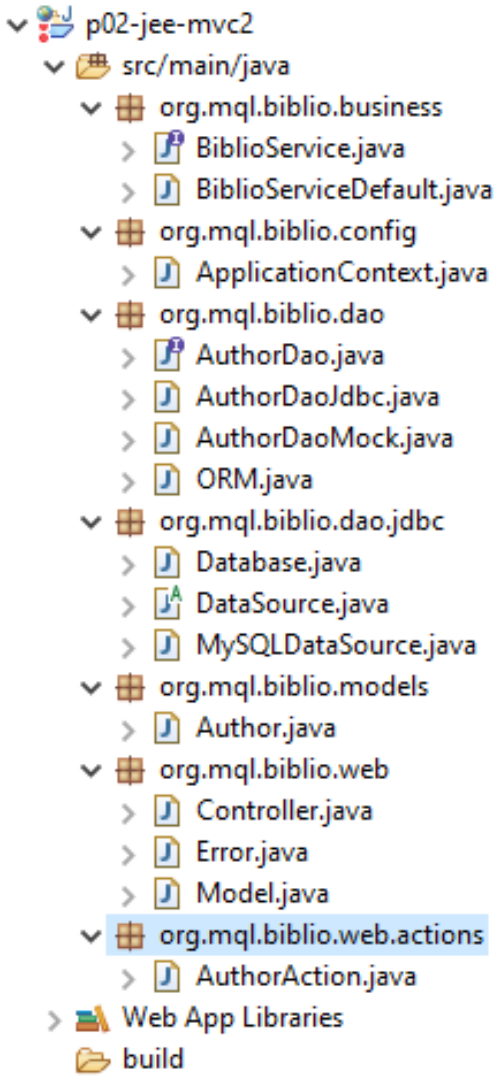

FIGURE 1 – architecture général

— **Couche présentation (web)** :
  — `Controller.java` : Contrôleur principal qui traite les requêtes HTTP.
  — `AuthorAction.java` : Classe d'action qui appelle la couche métier.
  — `Model.java`, `Error.java` : Objets utilisés pour transférer les données vers les vues.

  - **Vues JSP** : Affichent les données (non visibles ici, mais appelées via suffixe `/views/*.jsp`).
- **Couche métier (service)** :
  - `BiblioService.java` : Interface déclarant les opérations métier.
  - `BiblioServiceDefault.java` : Implémentation concrète du service.
- **Couche DAO (Data Access Object)** :
  - `AuthorDao.java` : Interface DAO.
  - `AuthorDaoJdbc.java`, `AuthorDaoMock.java` : Implémentations concrètes.
  - `ORM.java`, `Database.java`, `MySQLDataSource.java` : Classes techniques d'accès à la base.
- **Configuration** :
  - `ApplicationContext.java` : Classe de configuration qui instancie les composants (équivalent d'un conteneur IoC).

1.2 Interface `BiblioService`

L'interface `BiblioService` définit les opérations métier sur les auteurs. Elle permet d'avoir plusieurs implémentations sans modifier le code client, respectant ainsi le principe **Open/Closed** du SOLID.

1.3 `BiblioServiceDefault` implémente `BiblioService`

- L'attribut `AuthorDao` est injecté via le constructeur :

```
public BiblioServiceDefault(AuthorDao authorDao) {
    this.authorDao = authorDao;
}
```

- Le service est faiblement couplé à la couche DAO (il dépend de l'interface).
- Le code respecte le principe de l'inversion de dépendance.

1.4 `ApplicationContext` : Classe de configuration

Cette classe joue le rôle de fabrique de composants (beans), avec un pattern Singleton :

```
private static BiblioService biblioService = null;

public static BiblioService getBiblioService() {
    if (biblioService == null) {
        biblioService = new BiblioServiceDefault(new AuthorDaoJdbc());
    }
    return biblioService;
}
```

1.5 Interface `AuthorDao`

— Déclare les opérations CRUD (ex : `selectAll()`, `findById()`).
— Peut être implémentée par :
  — `AuthorDaoJdbc` : pour un accès réel à une base SQL.
  — `AuthorDaoMock` : pour des données simulées.

1.6 Couche Web – Controller et Actions

— `Controller.java` : Contrôleur central de l'application, qui route les requêtes.
— `AuthorAction.java` : Action associée à la gestion des auteurs.
— `Model.java` : Objet envoyé à la vue pour affichage.
— `Error.java` : Objet utilisé pour transmettre les messages d'erreur à la vue.
— Vues JSP appelées via un préfixe/suffixe : `/views/ + authors.jsp`.

Remarque : L'API JSON peut être implémentée avec `ObjectMapper` de Jackson, bien que ce ne soit pas natif en JEE.

2 Spring :

Dans le cadre de notre apprentissage du framework Spring, nous avons réalisé et étudié plusieurs projets Java en environnement Web et Desktop. Ce rapport vise à présenter une analyse technique approfondie de ces projets, en mettant en avant la structure, la configuration (annotations vs XML), les technologies utilisées, ainsi que les compétences mobilisées.

Objectifs :

- Comprendre les différences entre les configurations XML et par annotations dans Spring.
- Comparer les architectures des applications Web et Desktop.
- Identifier les avantages et inconvénients de chaque approche.

2.1 desktop

2.1.1 version xml :

Ce projet Spring est une application console (desktop) qui utilise une configuration XML classique via le fichier beans.xml pour déclarer les beans et gérer les dépendances via le conteneur Spring. Il s'agit d'une application simple permettant de manipuler une liste d'auteurs via un service métier injecté.

1. Architecture général du projet :

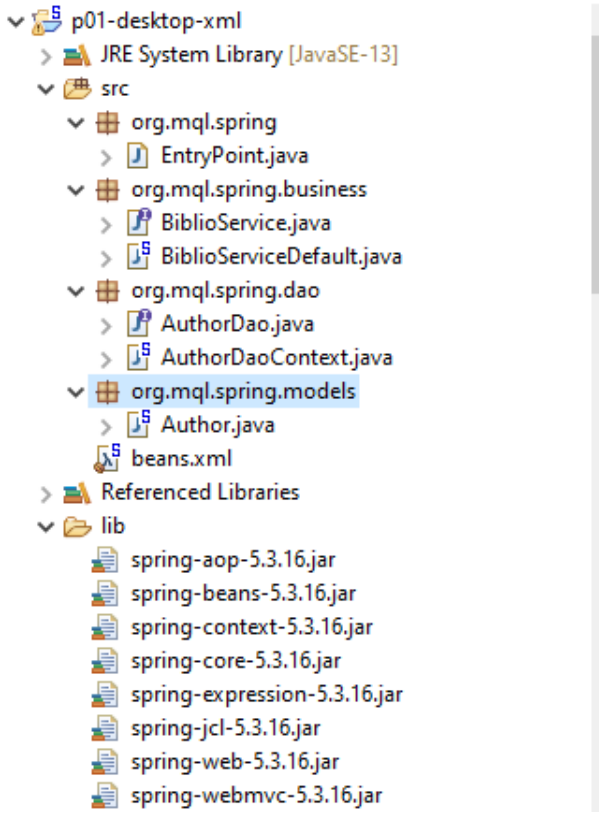

FIGURE 2 – architecture général

a. Structure générale :

- src/ : Contient tout le code source Java organisé par packages fonctionnels.

- lib/ : contient l'ensemble des bibliothèques externes nécessaires au fonctionnement d'un projet Spring. Ces fichiers .jar permettent de fournir les fonctionnalités essentielles du framework :

- la gestion des beans (spring-beans)
- l'injection de dépendances et le conteneur IoC (spring-core, spring-context)
- la prise en charge des requêtes HTTP et du modèle MVC (spring-web, spring-webmvc).

b. Détail des packages :

- org.mql.spring :

- EntryPoint.java : classe principale contenant la méthode main(). Elle initialise le contexte Spring et exécute des scénarios (exp01(), exp02()).

- org.mql.spring.business :

- BiblioService : interface définissant les opérations métier.

- BiblioServiceDefault : implémentation concrète avec injection de AuthorDao via un setter.

- org.mql.spring.dao :

- AuthorDao : interface de la couche DAO.

- AuthorDaoContext : implémentation contenant une liste simulée d'auteurs.

- org.mql.spring.models :

-Author : classe POJO représentant un auteur avec id, name, yearBorn.

- beans.xml : : fichier de configuration Spring contenant la déclaration manuelle des beans.

2. Configuration XML (beans.xml) Le fichier beans.xml configure :

- Le scan automatique des annotations avec :

```
<context:component-scan base-package="org.mql.spring"/>
```

- L'activation de l'injection via annotations (@Repository, @Component, etc.) :

```
<context:annotation-config/>
```

- La déclaration manuelle de deux beans de type Author :

```
<bean id="a01" class="org.mql.spring.models.Author">
  <property name="id" value="101"/>
  <property name="name" value="Rod Johnson"/>
  <property name="yearBorn" value="1970"/>
</bean>
```

- Le bean BiblioServiceDefault, instancié sans injection explicite dans le XML, mais compatible avec l'injection automatique byType grâce à l'attribut :

```
<beans default-autowire="byType">
```

3. Fonctionnement de l'application

3.1 La classe EntryPoint charge le contexte Spring :

```
        context = new ClassPathXmlApplicationContext("beans.xml");
```

- Deux méthodes de test :

- exp01() : récupère plusieurs fois le bean "a01" pour vérifier son cycle de vie (singleton).

```
void exp01() {
    Author a1 = context.getBean("a01", Author.class);
    System.out.println("a1 =" + a1);

    Author a2 = context.getBean("a01", Author.class);
    System.out.println("a2 =" + a2);

    Author a3 = context.getBean("a01", Author.class);
    System.out.println("a3 =" + a3);
}
```

- exp02() : utilise le BiblioService pour afficher la liste des auteurs.

```
void exp02() {
    BiblioService service = context.getBean(BiblioService.class);
    List<Author> authors =  service.getAllAuthors();
    System.out.println(authors);
}
```

3.2 Injection de dépendance : Injection de dépendances par setter (dans BiblioServiceDefault et AuthorDaoContext) :

```
public void setAuthorDao(AuthorDao authorDao) { ... }
public void setAuthor(List<Author> authors) { ... }
```

Le projet utilise default-autowire="byType", ce qui permet à Spring d'injecter automatiquement les beans compatibles sans avoir besoin d'écrire les <property> dans le XML.

2.1.2 version annotations :

Ce projet Spring illustre une application desktop (console) développée avec une configuration 100 100% Java, c'est-à-dire sans fichier XML. Tous les composants Spring sont configurés grâce aux annotations @Configuration, @Bean, @Service, @Repository et @Autowired.

1. **Architecture général du projet :**

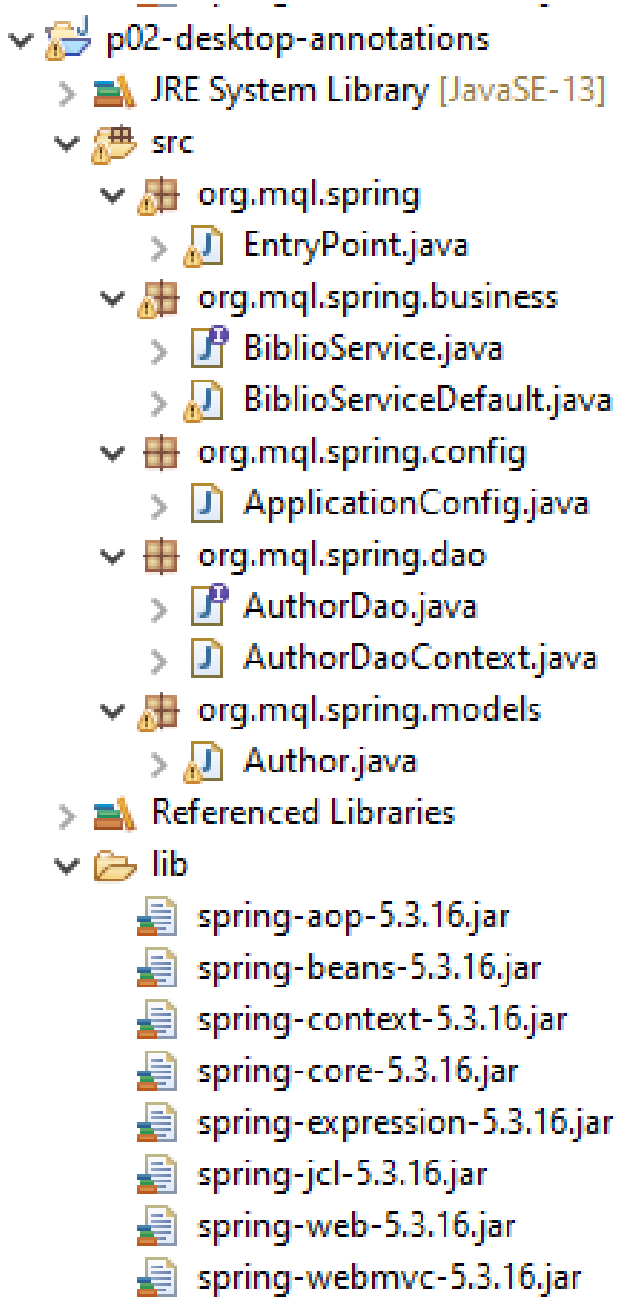

FIGURE 3 – architecture général

a. Structure générale :

- src/ : Contient tout le code source Java organisé par packages fonctionnels.

- lib/ : contient les bibliothèques Spring nécessaires (spring-beans, spring-core, spring-context, etc.).

b.Détail des packages :

- org.mql.spring :

- EntryPoint.java : classe principale avec la méthode main(). Elle initialise manuellement un contexte AnnotationConfigApplicationContext et exécute des scénarios métier.

- org.mql.spring.business :

- BiblioService : interface définissant les opérations métier.

- BiblioServiceDefault : classe annotée avec @Service et recevant le DAO via injection par constructeur.

- org.mql.spring.dao :

- AuthorDao : interface de la couche DAO.

- AuthorDaoContext : implémentation annotée avec @Repository, recevant la liste des auteurs via @Autowired (injection directe dans l'attribut).

- org.mql.spring.models :

-Author : classe POJO représentant un auteur avec id, name, yearBorn.

- org.mql.spring.config :

- ApplicationConfig : classe annotée @Configuration définissant plusieurs beans manuellement à l'aide de @Bean.

1. Fonctionnement de l'application :

a- Point d'entrée du projet : EntryPoint.java La classe EntryPoint représente le point d'entrée principal du projet desktop utilisant Spring avec configuration par annotations.

Elle initialise le contexte Spring à partir de la classe de configuration ApplicationConfig, puis exécute deux expériences :

- exp01() : démonstration du bean singleton.

```
void exp01() {
  Author a1 = context.getBean("a01", Author.class);
  System.out.println("a1 =" + a1);
  Author a2 = context.getBean("a01", Author.class);
  System.out.println("a2 =" + a2);
  Author a3 = context.getBean("a01", Author.class);
  System.out.println("a3 =" + a3);
}
```

- exp02() : affichage du titre et des auteurs à l'aide du service métier

```
System.out.println(context.getBean(String.class));
BiblioService service = context.getBean(BiblioService.class);
List<Author> authors = service.getAllAuthors();
```

Le service est injecté automatiquement avec son DAO, qui reçoit lui-même la liste d'auteurs grâce à l'injection par attribut.

Initialise un contexte Spring à partir d'une configuration Java-based (sans XML).

```
context = new AnnotationConfigApplicationContext(ApplicationConfig.
    class);
```

b- Explication des injections :

- @Autowired sur attribut (AuthorDaoContext) :

```
@Autowired
private List<Author> authors;
```

Spring injecte directement un bean de type List<Author> via réflexion, sans besoin de setter.

Important : Il faut déclarer ce bean dans ApplicationConfig

```
@Bean
public List<Author> authors() {
    return Arrays.asList(a01(), a02());
}
```

- Injection par constructeur (BiblioServiceDefault) :

```
public BiblioServiceDefault(AuthorDao authorDao) {
    this.authorDao = authorDao;
}
```

Grâce à l'annotation @Service, Spring instancie cette classe et injecte automatiquement le DAO via son seul constructeur.

Ce projet montre une bonne maîtrise de Spring par annotations avec une configuration claire et moderne :

- Utilisation du conteneur Spring via AnnotationConfigApplicationContext

- Utilisation des annotations @Configuration, @Bean, @ComponentScan, @Service, @Repository, @Autowired

- Respect de l'architecture en couches : Modèle, DAO, Service, Entrée

2.1.3 Comparaison entre la configuration XML et Annotations dans un projet Spring desktop :

Critère	**Version XML**	**Version Annotations (Java-based)**
Type de configuration	Déclaration manuelle via `beans.xml`.	Configuration 100% Java via la classe `ApplicationConfig.java`.
Point d'entrée	`ClassPathXmlApplicationContext` pour charger le fichier XML.	`AnnotationConfigApplicationContext` pour charger la config Java.
Déclaration des beans	Utilise les balises `<bean>` dans le fichier XML.	Utilise l'annotation `@Bean` dans une classe annotée `@Configuration`.
Injection de dépendances	Par setter ou via `default-autowire="byType"`.	Par constructeur ou champ avec `@Autowired`.
Gestion des dépendances	Centralisée dans un seul fichier, peu flexible.	Locale et modulaire via les annotations, plus souple.
Lisibilité / Maintenance	Peut devenir illisible sur de gros projets.	Plus clair, modulaire, facilement maintenable.
Souplesse / Réutilisabilité	Moins souple, difficile à adapter sans modifier XML.	Plus souple, changement de dépendance facile dans le code.
Structure technique	`beans.xml` + `EntryPoint.java`.	Tout est intégré dans le code Java.
Complexité de configuration	Verbeuse, lourde à maintenir.	Légère, intuitive, moderne.
Erreurs fréquentes	Typage non vérifié à la compilation.	Détection d'erreurs à la compilation (type-safe).
Apprentissage	Utile pour bien comprendre le fonctionnement de Spring.	Préférée pour les projets professionnels modernes.

TABLE 1 – Comparaison entre la configuration XML et Annotations dans Spring Desktop

2.2 web :

Architecture générale :

Dans le cadre de notre apprentissage approfondi du framework Spring, nous avons mené plusieurs projets Java en environnement Web, en complément de ceux réalisés en mode desktop. Ce rapport a pour objectif de présenter une analyse technique détaillée de ces projets web, en mettant en évidence la structure des applications, les différentes approches de configuration (XML vs Annotations), les technologies web utilisées (Spring MVC, JSP, Servlet, etc.), ainsi que les compétences clés mobilisées durant leur réalisation.

2.2.1 Projet 1 : Configuration par XML :

1. Architecture du projet p03-web-xml :

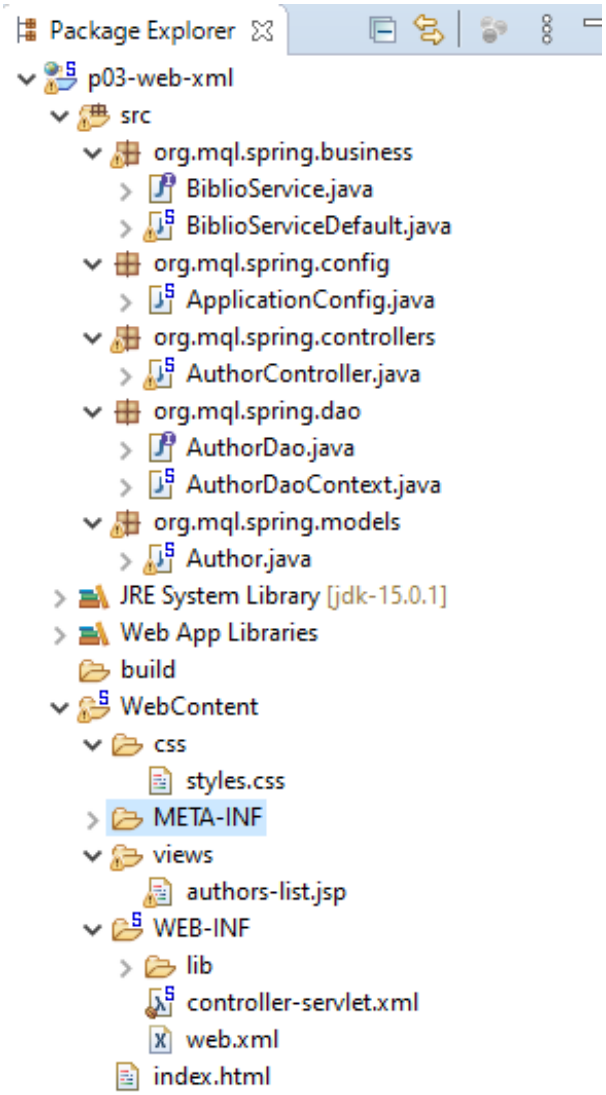

FIGURE 4 – architecture général

a. Structure générale :

- src/ : Contient tout le code source Java organisé par packages fonctionnels.

- WebContent/ : Contient les fichiers de ressources web (HTML, CSS, JSP) ainsi que la configuration web (web.xml).

b. Détail des packages :

- org.mql.spring.business : Contient la logique métier de l'application :

- BiblioService : Interface du service métier.

- BiblioServiceDefault Implémentation concrète du service.

- org.mql.spring.config : Contient la configuration Spring en Java :

- ApplicationConfig : Classe de configuration de l'application.

- org.mql.spring.controllers : Contient les contrôleurs Spring MVC :

- AuthorController : Gère les requêtes liées aux auteurs, communique avec le service.

- org.mql.spring.dao : Contient la couche accès aux données (DAO) :

- AuthorDao : Interface pour les opérations CRUD sur les auteurs.

- AuthorDaoContext : Implémentation concrète avec une source de données simulée.

- org.mql.spring.models : Contient les modèles de données (DTO/POJO)

- Author : Classe représentant un auteur avec ses attributs.

c. Répertoire WebContent :

- css/ :

- styles.css : Feuille de style utilisée pour le front-end.

- views/ :

- authors-list.jsp : : Vue JSP affichant la liste des auteurs.

- WEB-INF/ :

- controller-servlet.xml : Fichier de configuration de Spring MVC (mapping des beans, viewResolver, etc.).

- web.xml : Descripteur de déploiement, configure le DispatcherServlet de Spring.

-index.html : Page d'accueil de l'application (optionnelle ou utilisée pour redirection).

2. Initialisation : Utilisation du fichier web.xml

Le fichier web.xml, également appelé Deployment Descriptor, est un fichier standard de Java EE qui permet de configurer le comportement de l'application web. Il joue un rôle central dans l'initialisation de l'environnement servlet.

Rôle principal :

- Configurer le conteneur web (Tomcat, Jetty...)
- Définir comment les requêtes HTTP sont routées
- Déclarer les servlets, leurs mappings et paramètres initiaux

- Configurer la sécurité, les pages d'erreur, les encodages, etc.
- Définir les fichiers de bienvenue via la balise <welcome-file-list>

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

Cela permet d'afficher automatiquement l'un de ces fichiers lorsqu'un utilisateur accède à la racine du projet.

- définir les requêtes HTTP prises en charge par cette servlet

```
<servlet-mapping>
  <servlet-name>controller</servlet-name>
  <url-pattern>/biblio/*</url-pattern>
</servlet-mapping>
```

- Déclaration du DispatcherServlet de Spring :

Dans une application Spring MVC, le DispatcherServlet est la pièce centrale du framework. C'est un Front Controller qui intercepte toutes les requêtes entrantes et les délègue aux composants Spring appropriés (contrôleurs, vues, etc.).

Voici comment il est déclaré dans le fichier web.xml :

```
<servlet>
  <servlet-name>controller</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</
  servlet-class>
</servlet>
```

- <servlet-name> : nom logique de la servlet, ici controller.

- <servlet-class> : classe de la servlet Spring.

- Fonctionnement du DispatcherServlet :

1. Réception de toutes les requêtes :

Exemple : un utilisateur accède à /biblio/authors-list

2. Délégation :

Le DispatcherServlet interroge les HandlerMappings pour identifier le contrôleur Spring correspondant.

3. Traitement : Dans notre cas, la requête est dirigée vers la méthode authorsList() dans AuthorController, grâce à l'annotation :

```
@GetMapping("authors-list")
```

4. Résolution de la vue : Le contrôleur retourne un nom de vue et un modèle, que le DispatcherServlet transmet au ViewResolver pour générer la réponse (ex. JSP).

5. Réponse : La page (ex. authors-list.jsp) est affichée avec les données fournies par le contrôleur.

3. Configuration :

La configuration de Spring MVC est assurée via le fichier XML controller-servlet.xml, déclaré dans le descripteur de déploiement web.xml. Ce fichier joue un rôle fondamental dans la détection automatique des composants Spring, notamment les contrôleurs.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <context:component-scan base-package="org.mql.spring" />
</beans>
```

Rôle de controller-servlet.xml :

- Activer la détection automatique des composants grâce à <context :component-scan>
- Permettre à Spring MVC de gérer les contrôleurs annotés avec @Controller
- Faciliter l'injection de dépendances entre beans via le conteneur Spring

À noter : ce fichier est chargé automatiquement par le DispatcherServlet déclaré dans web.xml, et spécifiquement associé à la couche web (il ne remplace pas l'éventuel applicationContext.xml utilisé pour la logique métier ou la persistance).

Le fichier controller-servlet.xml permet de configurer la partie web de l'application Spring MVC. Grâce à la balise <context :component-scan>, il active le scannage automatique des classes Java annotées comme @Controller, @Service ou @Repository. Ce fichier est directement chargé par le DispatcherServlet, défini dans le fichier web.xml.

5. Injection de dépendances en XML : Spring permet d'injecter les dépendances entre objets (beans) via le fichier de configuration XML. Cela permet de définir qui dépend de quoi, sans modifier le code source. Il existe plusieurs façons d'injecter une dépendance :

1. Injection par constructeur :

```
        <bean id="service" class="org.mql.spring.business.
    BiblioServiceDefault">
            <constructor-arg ref="authorDao"/>
        </bean>
```

Ici, l'instance de BiblioServiceDefault reçoit authorDao via son constructeur.

2. Injection par setter : Utilisée quand la dépendance peut être injectée après la création du bean.

```
<bean id="authorDao" class="org.mql.spring.dao.AuthorDaoContext">
  <property name="authors" ref="listAuthors" />
</bean>
```

Le setter setAuthors() sera automatiquement appelé avec la collection listAuthors.

3. Injection de valeurs simples : Utilisée quand la dépendance peut être injectée après la création du bean.

```
  <bean id= "a02" class="org.mql.spring.models.Author">
    <property name="id" value="102"/>
```

```
    <property name="name" value="Erich Gamma"/>
    <property name="yearBorn" value="1961"/>
  </bean>

```

Ce bean représente une simple chaîne de texte.

4. Injection de collections : Permet d'injecter des listes, ensembles ou maps de beans.

```
<bean id="listAuthors" class="java.util.ArrayList">
  <constructor-arg>
    <list>
      <ref bean="a01" />
      <ref bean="a02" />
    </list>
  </constructor-arg>
</bean>
```

Cette liste sera injectée là où un List<Author> est attendu.

Avantages de l'injection XML

- Configuration centralisée : Tout est défini dans un seul fichier XML, ce qui améliore la lisibilité.

- Souplesse : On peut changer les dépendances sans toucher au code source.

- Adapté aux environnements multi-configuration : Idéal pour des déploiements avec des dépendances différentes selon l'environnement (dev/test/prod).

2.2.2 Projet 2 : Configuration par Annotations :

1. Architecture du projet p04-annotation-xml :

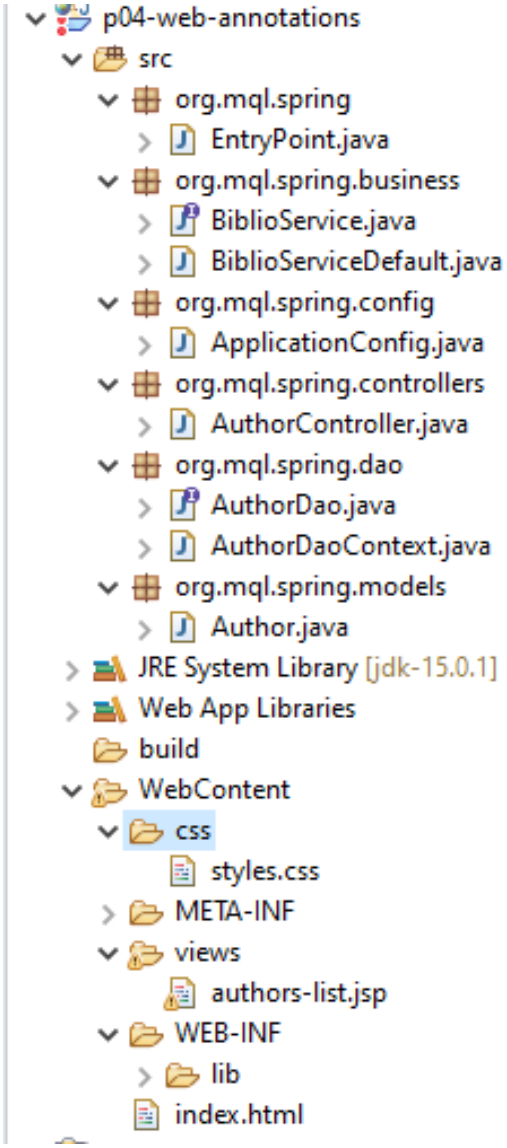

FIGURE 5 – architecture général

a. Structure générale :

- src/ : Contient tout le code source Java organisé par packages fonctionnels.

- WebContent/ : Contient les fichiers de ressources web (HTML, CSS, JSP) ainsi que la configuration web (web.xml).

b. Détail des packages :

- org.mql.spring : EntryPoint.java

Dans le cadre d'une configuration entièrement basée sur les annotations (sans fichier XML), la configuration du DispatcherServlet n'est pas réalisée via un fichier web.xml, mais à travers une classe Java dédiée : EntryPoint. Celle-ci implémente l'interface WebApplicationInitializer, ce qui permet à Spring de

détecter et de lancer automatiquement la configuration de l'application. Elle crée un contexte Spring basé sur la classe ApplicationConfig, puis enregistre dynamiquement le DispatcherServlet, avec un mapping sur l'URL /biblio/*. Cela illustre l'approche moderne de Spring basée sur le Java pur, sans XML.

- Création d'un contexte Spring basé sur les annotations : Interface du service métier.

```
AnnotationConfigWebApplicationContext context = new
    AnnotationConfigWebApplicationContext();
```

- Enregistrement de la classe de configuration Spring

```
        context.register(ApplicationConfig.class);
```

- Déclaration et enregistrement du DispatcherServlet

```
Dynamic controller = servletContext.addServlet("controller",
    new DispatcherServlet(context));
```

- Mapping de l'URL des requêtes entrantes

```
    controller.addMapping("/biblio/*");
```

- Priorité de chargement au démarrage

```
    controller.setLoadOnStartup(1);
```

- org.mql.spring.business : Contient la logique métier de l'application :

- BiblioService : Interface du service métier.

- BiblioServiceDefault Implémentation concrète du service.

- org.mql.spring.config : Contient la configuration Spring en Java :

- ApplicationConfig : Classe annotée avec @Configuration qui déclare les beans et configure le scan des composants.

- org.mql.spring.controllers : Contient les contrôleurs Spring MVC :

- AuthorController : Gère les requêtes liées aux auteurs, communique avec le service.

- org.mql.spring.dao : Contient la couche accès aux données (DAO) :

- AuthorDao : Interface pour les opérations CRUD sur les auteurs.

- AuthorDaoContext : Implémentation concrète avec une source de données simulée.

- org.mql.spring.models : Contient les modèles de données (DTO/POJO)

- Author : Classe représentant un auteur avec ses attributs.

c. Répertoire WebContent :

- css/ :

- styles.css : Feuille de style utilisée pour le front-end.

- views/ :

- authors-list.jsp : : Vue JSP affichant la liste des auteurs.

-index.html : La page d'accueil de l'application (index.html) contient un lien vers la route biblio/authors-list. C'est une bonne pratique pour tester rapidement les fonctionnalités de navigation.

Vérifications à effectuer : Le contrôleur AuthorController doit contenir une méthode avec l'annotation :

@GetMapping("authors-list") Le fichier authors-list.jsp (présent dans WebContent/views/) doit être correctement résolu par le ViewResolver de Spring MVC.

1. Configuration Java-based : ApplicationConfig.java : La classe ApplicationConfig est bien définie avec :

-@Configuration indique que cette classe contient des définitions de beans.

-@ComponentScan("org.mql.spring") permet à Spring de scanner automatiquement les classes annotées (@Service, @Repository, @Controller).

- declaration beans avec @Bean : Dans le cadre de la configuration Java-based, il est possible de déclarer manuellement des beans à l'aide de l'annotation @Bean, à l'intérieur d'une classe annotée avec @Configuration. Cela permet à Spring d'instancier ces objets et de les gérer dans son conteneur IoC (Inversion of Control).

Dans ce projet, trois beans sont définis explicitement dans la classe ApplicationConfig

```
@Bean
public Author a01() {
  return new Author(101, "Rod Johnson", 2003);
}

@Bean
public Author a02() {
  return new Author(102, "Erich Gamma", 1961);
}

@Bean
public String title() {
  return "Gestion de bibliotheque";
}

```

a01() et a02() : créent deux objets de type Author, représentant des auteurs fictifs. Ces beans peuvent ensuite être injectés dans d'autres composants (DAO, services, etc.).

title() : fournit une chaîne de caractères utilisée dans l'application, par exemple comme titre général ou entête dans l'interface utilisateur.

1. Injection de dépendances : :

- **Injection via constructeur (Service) :** Dans la classe BiblioServiceDefault, l'injection de dépendances est réalisée implicitement par constructeur, car la classe ne possède qu'un seul constructeur avec un paramètre. Spring peut donc déduire automatiquement quel bean injecter.

```java
@Service
public class BiblioServiceDefault implements BiblioService {
    private AuthorDao authorDao;

    public BiblioServiceDefault(AuthorDao authorDao) {
        this.authorDao = authorDao;
    }
}
```

- **injection par champ** Dans la classe AuthorDaoContext, une injection par champ est utilisée :

```java
@Autowired
private List<Author> authors;

```

Cette injection fonctionne même sans méthode setter, car Spring utilise la réflexion Java pour injecter directement la dépendance dans l'attribut privé.

Remarque importante : Pour que cette injection fonctionne, il faut qu'un bean de type List<Author> soit déclaré dans la configuration. Sinon, une erreur sera levée à l'exécution.

Voici le bean à ajouter dans ApplicationConfig :

```java

@Bean
public List<Author> authors() {
    return Arrays.asList(a01(), a02());
}

```

Le projet démontre une bonne maîtrise de Spring avec annotations, en séparant clairement les responsabilités. L'utilisation de @Bean, @Service, @Repository, @Autowired, combinée au @ComponentScan, permet une gestion claire, souple et moderne des dépendances.

2.2.3 Comparaison entre la configuration XML et Annotations dans un projet Spring Web :

Critère	**Version XML (web.xml + controller-servlet.xml)**	**Version Annotations (Java-based)**
Initialisation	Via `web.xml` (Deployment Descriptor)	Via classe Java qui implémente `WebApplicationInitializer`
Configuration Spring	Déclaration des beans et scan dans `controller-servlet.xml`	Classe `ApplicationConfig.java` avec `@Configuration` et `@ComponentScan`
Déclaration du DispatcherServlet	Dans `web.xml`, via `<servlet>` et `<servlet-mapping>`	Programmatiquement via `Dynamic controller = servletContext.addServlet(...)`
Gestion des dépendances	Injection via `<constructor-arg>`, `<property>`, ou collections dans XML	Injection via `@Autowired` (champ ou constructeur)
Détection des composants	Activée via `<context:component-scan>` dans XML	Activée via `@ComponentScan("package")` dans la config Java
Déclaration des Beans	Avec `<bean>` dans le XML	Avec `@Bean` dans la classe de configuration
Contrôleurs Spring MVC	Annotés avec `@Controller`, détectés via le XML	Identiques, mais détectés via le scan automatique Java
Mapping des requêtes	Géré par le `DispatcherServlet` déclaré dans XML	Géré par le `DispatcherServlet` enregistré dans le code
Gestion des vues JSP	Définie dans `controller-servlet.xml` via un `InternalResourceViewResolver`	Définie dans la classe config via un `@Bean` de type `ViewResolver`
Souplesse / Maintenance	Moins souple, nécessite modification du XML	Plus flexible, tout est dans le code, maintenable facilement
Utilisation professionnelle	Approche plus ancienne, encore utilisée dans certains contextes legacy	Recommandée pour les projets modernes Spring Boot / MVC

TABLE 2 – Comparaison entre la configuration XML et Annotations dans un projet Spring Web

3 Spring-boot :

3.1 Spring Boot avec JPA :

1. P07SpringBootJpaTestApplication.java :

- Rôle : C'est la classe principale qui démarre l'application Spring Boot.

- Détails : @SpringBootApplication Active automatiquement :

@Configuration : pour déclarer des beans

@ComponentScan : pour scanner les composants

@EnableAutoConfiguration : pour configurer automatiquement Spring

-@Bean public Author a03() : Déclare deux beans Author (a03 et a04) injectés plus tard par Spring (utilisé dans AuthorDaoContext).

@Bean public String title() : Injecte une chaîne "Gestion de bibliotheque" utilisée dans le contrôleur MVC (BiblioController).

2. Author.java :

- Rôle : C'est une entité JPA, qui représente un auteur dans la base de données.

- Annotations :

@Entity : déclare une entité JPA

@Table(name="Authors") : lie cette classe à la table Authors

@Id :clé primaire

@Column(name= "AuID") : mappe l'attribut à une colonne SQL spécifique

3. AuthorRepository.java :

- Rôle : Interface Spring Data JPA qui permet d'accéder à la base sans implémentation manuelle.

- Héritage : Hérite de toutes les méthodes CRUD (findAll, save, delete, etc.).

- Méthodes personnalisées : List<Author> findByName(String name); Recherche automatique par nom (Spring Data Query Derivation)

- Recherche automatique par nom (Spring Data Query Derivation) : Utilise **JPQL** pour faire une recherche par mot-clé.

- BiblioService.java : C'est une interface définissant les fonctions métier.

4. BiblioServiceDefault.java :

Implémentation manuelle du service métier, qui utilise un DAO AuthorDao.

public BiblioServiceDefault(AuthorDao authorDao) Injection implicite de dépendance (par constructeur)

5. BiblioServiceJpa.java : Implémentation automatisée du service métier en utilisant Spring Data JPA (AuthorRepository).

- Annotations : @Service Déclare cette classe comme composant métier (Service)

6. AuthorRestController.java : Expose les services d'auteurs via une API RESTful (GET).

- Annotations :

@RestController @RequestMapping("/api/authors") Les endpoints commencent tous par /api/authors

- Méthodes :

-/api/authors → getAllAuthors()

-/api/authors/id → getAuthorById() avec gestion 404 via ResponseEntity

-/api/authors ?keyword=sa → recherche par mot-clé avec @RequestParam

7. BiblioController.java :

- Rôle : Contrôleur MVC Spring (interface utilisateur via Thymeleaf ou JSP).

- Annotations :

@Controller @RequestMapping("/biblio") Les endpoints commencent tous par /api/authors

- Méthodes :

- /biblio/authorslist : ajoute les auteurs + le titre au Model et retourne "authors-list" (vue).

3.2 Annotations présentes dans ton code :

@RestController :

— **But :**Indique que la classe est un contrôleur REST. Elle combine" :

- @Controller : pour identifier une classe comme contrôleur.

- @ResponseBody : pour indiquer que les méthodes retournent directement les données (JSON, XML...) au lieu d'une vue.

@RequestMapping :

- But : Spécifie le chemin de base d'une requête HTTP pour une classe ou une méthode.

- Utilisation :

En classe → chemin racine.

En méthode → chemin spécifique.

@GetMapping :

- But :Gère une requête HTTP GET.

- Remplace :@RequestMapping(method = RequestMethod.GET)

@PathVariable :

- But : Extrait une valeur dynamique de l'URL (ex : /api/authors/101) → récupère 101

@RequestParam :

- But : Extrait un paramètre de la requête (ex : /api/authors ?keyword=saida)

@Autowired :

- But : Injection automatique des dépendances par le conteneur Spring

@Service

- But : Indique que la classe représente une couche métier (logique de l'application).

@Repository

- But : Indique que la classe est une couche d'accès aux données (DAO).

Spring gère aussi les exceptions SQL automatiquement.

1. @Autowired private BiblioService service ; Spring voit que tu veux une instance d'un bean de type BiblioService. Comme il existe une seule classe qui l'implémente et qui est annotée avec @Service (BiblioServiceJpa), il l'injecte automatiquement ici.

L'annotation @Service fait que Spring crée un objet (bean) de cette classe et le garde dans son contexte d'application.

C'est comme si tu faisais : BiblioService service = new BiblioServiceJpa() ;

Ton scénario résumé Tu as :

1. Une interface :

public interface BiblioService // méthodes 2. Une implémentation annotée avec @Service :

@Service public class BiblioServiceJpa implements BiblioService // implémentation des méthodes 3. Une injection dans un contrôleur :

@Autowired private BiblioService service ;

Ce que Spring fait automatiquement : Au démarrage de l'application, Spring scanne les classes avec @Component, @Service, @Repository, @Controller, etc.

Il voit @Service sur BiblioServiceJpa Spring instancie cette classe et l'enregistre comme bean dans son container.

Quand Spring voit @Autowired BiblioService service ;, il cherche un bean qui implémente l'interface BiblioService.

Il trouve BiblioServiceJpa, et il fait automatiquement :

this.service = new BiblioServiceJpa() ; // (mais sans "new", c'est le container Spring qui gère)

Résultat : tu peux appeler service.getAllAuthors() et ça exécutera la méthode dans BiblioServiceJpa.

Tu peux donc dire : Oui, @Autowired BiblioService service signifie : "Spring, injecte-moi une instance d'une classe qui implémente BiblioService."

Et comme tu as annoté BiblioServiceJpa avec @Service, Spring sait que c'est le bon candidat.

Si tu avais plusieurs implémentations de BiblioService Par exemple :

@Service public class BiblioServiceJpa implements BiblioService

@Service public class BiblioServiceDefault implements BiblioService Là, Spring ne saura pas laquelle injecter, donc tu devras préciser avec :

@Qualifier("biblioServiceJpa") @Autowired private BiblioService service ; Ici, Spring va injecter le bean nommé biblioServiceJpa (le nom du bean est par défaut le nom de la classe avec la première lettre en minuscule).

Ou marquer une des deux classes avec @Primary. @Service @Primary // C'est celle-ci que Spring injectera par défaut public class BiblioServiceJpa implements BiblioService ...

En résumé Élément Ce que fait Spring @Service Enregistre un bean dans le container @Autowired Injecte automatiquement un bean BiblioService (interface) Permet de coder avec abstraction BiblioServiceJpa Implémentation réelle injectée @Qualifier Aide à choisir si plusieurs implémentations existent

JpaRepository — C'est quoi exactement ? JpaRepository est une interface fournie par Spring Data JPA qui permet de faire facilement des opérations CRUD (Create, Read, Update, Delete) sur une base de données sans écrire de code SQL.

Définition

public interface AuthorRepository extends JpaRepository<Author, Integer> Ici :

Author est l'entité (classe qui représente une table).

Integer est le type de la clé primaire (ID de l'auteur).

Grâce à JpaRepository, tu n'écris aucune requête SQL, mais tu as déjà accès à : author-Repository.findAll() ; // SELECT * FROM authors authorRepository.findById(1) ; // SELECT * FROM authors WHERE id = 1 authorRepository.save(author) ; // INSERT ou UPDATE authorRepository.deleteById(1) ; // DELETE FROM authors WHERE id = 1

Et plus encore si tu ajoutes des méthodes personnalisées comme :

List<Author> findByName(String name) ; // auto-génère SELECT * FROM authors WHERE name = ?

Quelle interface hériter ? JpaRepository hérite déjà de plusieurs autres interfaces :

JpaRepository PagingAndSortingRepository CrudRepository

Donc elle contient :

Méthodes de base (save, findAll, delete, etc.)

Pagination et tri

Personnalisation avec noms de méthodes

En résumé :

JpaRepository Interface Spring pour les opérations sur la base de données Avantages CRUD automatique, requêtes dynamiques, pagination Nécessite Une entité (@Entity) et une clé primaire (@Id)

application.properties : configuration de db

@Transient — C'est quoi ? L'annotation @Transient est utilisée en JPA (Java Persistence API) pour indiquer à l'ORM (comme Hibernate) d'ignorer un attribut lorsqu'il mappe une entité à une base de données.

Autrement dit :

Une variable marquée @Transient ne sera pas enregistrée dans la base de données.

many to one documents a publisher publisher a pls documents

. @ManyToOne + @JoinColumn — Relation avec Publisher java Copier Modifier @ManyToOne @JoinColumn(name = "Publisher$_ID$")$privatePublisherpublisher; Celaveutdire$:

Chaque Document est publié par un seul Publisher (éditeur).

La base contient une colonne Publisher$_IDdanslatableDocumentspourfairelaliaison.$

Donc : Plusieurs documents peuvent avoir le même Publisher → relation ManyToOne.

la semantique ? ? mecaniquement list des authors dans doc et pasd e sens de mettre list doc dans author mecaniquement marche semantiquement pas .

diff entre join table et joinColumns(de type annotation)

l avantage de json, xml db representation hearchique , mais pas dans le relationnel BD noSQL

p08 prj , jpa hibernate : on cree les models de donnees et les tables de la BD seront cree par jpa or hibernate

etapes de creation jpa prj : 1. pom.xml : starter springbootStarterjpa -connecteur 2. properties : jdbc params 3. models de donnees

comme impl de jpa on utilise hibernate spring.jpa.hibernate.ddl-auto=update <=> mise a jour

requete ddl create drop alter creation de donnees de test 1 cree data.sql file , va etre pris en consederation et les requettes va etre auto executee dans classpath obligatoirement et nommee data oblig

spring.sql.init.mode= fo9ach ghy execute le script never ola a chaque demarage always tous ce qui dml est dans data.sql

devtools redemarre l app auto

3.3 p05-spring-boot-jsp :

C'est un projet d'apprentissage, utile pour comprendre comment structurer, configurer et développer une application Spring Boot simple avec des vues JSP, tout en appliquant les bonnes pratiques de modularité et d'injection de dépendances.

Ce projet simule une mini-application de gestion des auteurs d'une bibliothèque. Il sert à :

- Afficher dynamiquement une liste d'auteurs sur une page JSP (authors-list.jsp).
- Fournir les mêmes données en format JSON via une API REST (/biblio/authors).
- Illustrer la manière dont Spring Boot automatise la configuration.
- Montrer comment les composants Spring interagissent (controllers, services, DAO, modèles).
- Préparer les étudiants/développeurs à créer des applications Spring plus avancées.

3.3.1 Structure du projet dans l'environnement de développement :

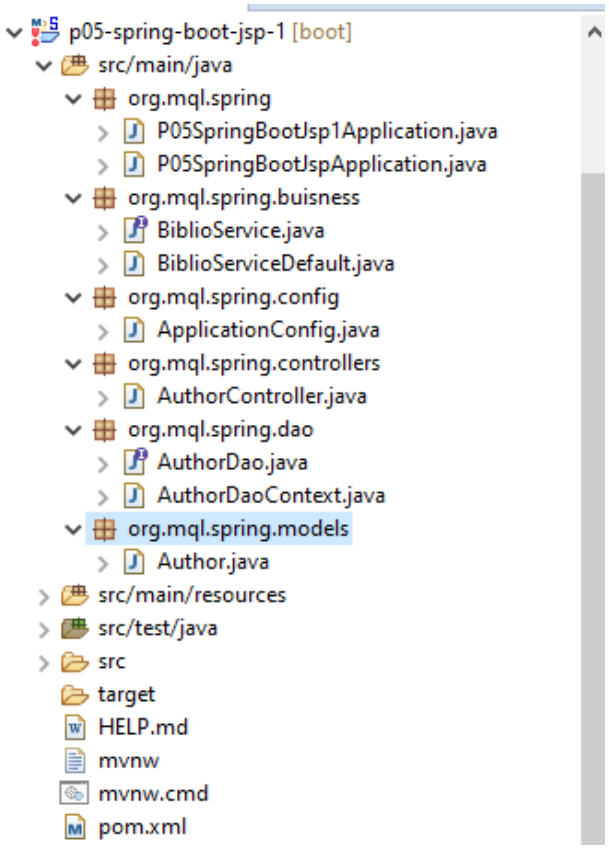

FIGURE 6 – architecture général

La figure ci-dessous montre la structure réelle du projet Spring Boot tel qu'organisé dans l'IDE (Eclipse). Cette architecture respecte les conventions standard de Spring Boot avec Maven.

Organisation des dossiers et fichiers :

— **src/main/java/org.mql.spring** : paquet racine du code source Java.

  — `business/` : contient l'interface `BiblioService` et son implémentation `BiblioServiceDefault`
  Cette couche représente la logique métier.
  — `config/` : contient la classe `ApplicationConfig` annotée `@Configuration`.
  Elle définit des `@Bean` manuellement (liste d'auteurs, titre).
  — `controllers/` : contient le contrôleur `AuthorController` qui gère les requêtes HTTP entrantes.
  — `dao/` : contient l'interface `AuthorDao` et la classe `AuthorDaoContext` annotée `@Repository` pour simuler l'accès aux données.
  — `models/` : contient le POJO `Author`, utilisé dans tout le projet pour représenter un auteur.

— **src/main/resources** : dossier contenant les ressources de l'application.

  — `application.properties` : fichier de configuration Spring.
  — `static/index.html` : page statique accessible via le navigateur à `/index.html`.
  — `templates/` : dossier destiné aux fichiers JSP (non visible ici, mais prévu par convention).

— **src/test/java** : contient les classes de test (vide pour l'instant).
— **target/** : dossier généré automatiquement à la compilation contenant le fichier `.jar`, les classes compilées et les rapports de test.
— **pom.xml** : fichier de configuration Maven, qui déclare les dépendances (comme `spring-boot-starter-web`) et les plugins.
— **mvnw, mvnw.cmd, .mvn/** : fichiers pour utiliser Maven Wrapper, permettant d'exécuter Maven sans l'installer globalement.
— **HELP.md** : fichier d'aide généré automatiquement à la création du projet (optionnel).

Remarques

— La structure est conforme aux bonnes pratiques de Spring Boot avec Maven.
— Chaque couche (modèle, DAO, service, contrôleur) est séparée logiquement.
— Les ressources sont placées dans `resources` conformément aux conventions.
— Cette organisation facilite la maintenance, les tests, et l'évolution du projet.

3.3.2 Le fichier pom.xml : la base de configuration :

Le fichier pom.xml est essentiel dans un projet Maven. Il définit :

- Le projet parent :

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.10</version>
</parent>
```

Cela permet d'hériter de la configuration de Spring Boot (gestion des versions, plugins...).

- Les dépendances utilisées :

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

Ajoute les composants nécessaires à une application web : Spring MVC, Tomcat embarqué, Jackson...

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

Fournit les outils pour écrire des tests unitaires (JUnit, Mockito...).

- Plugin Maven pour Spring Boot :

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

Permet d'exécuter le projet avec la commande mvn spring-boot :run.

3.3.3 Structure du projet et rôles des packages :

Voici une vue technique de chaque package du projet, avec les liaisons entre les composants Spring Boot.

- org.mql.spring : Contient la classe principale du projet :

```
@SpringBootApplication
public class P05SpringBootJspApplication {
    public static void main(String[] args) {
        SpringApplication.run(P05SpringBootJspApplication.class,
    args);
    }

    @Bean
    public Author a03() {
        return new Author(103, "Tim", 1955);
    }
}
```

Listing 1 – Classe principale avec `@SpringBootApplication`

L'annotation `@SpringBootApplication` combine :

— `@Configuration` : indique que la classe contient des définitions de beans.

— `@ComponentScan` : active la détection automatique des composants Spring.

— `@EnableAutoConfiguration` : active la configuration automatique de Spring Boot.

Le bean `Author` est injecté dans le contexte Spring, et peut être utilisé dans d'autres classes comme le DAO.

- org.mql.spring.models : Définit la classe métier `Author.java` :

```
public class Author {
    private int id;
    private String name;
    private int yearBorn;
    // Getters, setters, constructeur, toString...
}
```

C'est un POJO (Plain Old Java Object) représentant les auteurs. Il est utilisé dans tout le projet : contrôleur, service, DAO, et vue JSP.

- org.mql.spring.config : Contient la classe `ApplicationConfig.java`, annotée avec `@Configuration`. Elle déclare manuellement des beans Spring :

```
@Bean
public Author a01() {
    return new Author(101, "Rod Johnson", 1970);
}

@Bean
public Author a02() {
    return new Author(102, "Erich Gamma", 1961);
}

@Bean
public String title() {
    return "Gestion de biblioth que";
}
```

Ces beans sont injectés automatiquement grâce à `@Autowired`. Cette classe remplace les fichiers XML de configuration classiques.

Explication des annotations :

@Bean : Permet de déclarer manuellement un objet géré par Spring (un bean) dans une classe de configuration.

- org.mql.spring.dao :

Interface `AuthorDao.java` :

```
public interface AuthorDao {
    List<Author> selectAll();
}
```

Classe `AuthorDaoContext.java` :

```
@Repository
public class AuthorDaoContext implements AuthorDao {
    @Autowired
    private List<Author> authors;

```

```
    public List<Author> selectAll() {
        return authors;
    }
}
```

Cette classe simule l'accès aux données. La liste d'auteurs est injectée via les beans déclarés précédemment.

Explication des annotations :

@Repository : Marque une classe comme étant un composant d'accès aux données (DAO).

Permet à Spring de :

-la détecter automatiquement comme bean.

-lui attribuer un rôle lié à la persistance.

-gérer les exceptions spécifiques aux bases de données

- org.mql.spring.business :

Interface `BiblioService.java` :

```
public interface BiblioService {
    List<Author> getAllAuthors();
}
```

Classe `BiblioServiceDefault.java` :

```
@Service
public class BiblioServiceDefault implements BiblioService {
    private AuthorDao authorDao;

    public BiblioServiceDefault(AuthorDao authorDao) {
        this.authorDao = authorDao;
    }

    public List<Author> getAllAuthors() {
        return authorDao.selectAll();
    }
}
```

Le service interagit avec la couche DAO pour fournir les données métier au contrôleur.

Explication des annotations :

@Service : Indique que la classe contient la logique métier (business logic).
- Effet Comme @Component, elle permet l'injection automatique, mais donne une sémantique claire dans l'architecture : service métier.

- org.mql.spring.controllers :

Classe `AuthorController.java` :

```
@Controller
public class AuthorController {
    private BiblioService service;
```

```
4
5       @Autowired
6       private String title;
7
8       public AuthorController(BiblioService service) {
9           this.service = service;
10      }
11
12      @GetMapping("authors-list")
13      public String authorsList(Model model) {
14          List<Author> authors = service.getAllAuthors();
15          model.addAttribute("authors", authors);
16          model.addAttribute("title", title);
17          return "authors-list";
18      }
19
20      @GetMapping("/biblio/authors")
21      @ResponseBody
22      public List<Author> authors() {
23          return service.getAllAuthors();
24      }
25
26      @GetMapping("all-beans")
27      @ResponseBody
28      public String[] getAllBeans() {
29          return applicationContext.getBeanDefinitionNames();
30      }
31  }
```

Le contrôleur gère l'affichage de la vue JSP ainsi que l'API REST retournant la liste des auteurs au format JSON. **Explication des annotations dans la classe `AuthorController`**

La classe `AuthorController` utilise plusieurs annotations Spring qui permettent de gérer l'injection de dépendances, les routes HTTP, ainsi que la nature du contrôleur. Voici une analyse détaillée de chaque annotation utilisée :

- **`@Controller`** : Cette annotation indique que la classe est un contrôleur Spring MVC. Elle sera automatiquement détectée par le *component-scan* et enregistrée comme un bean gérant des requêtes HTTP. Elle permet de retourner des vues (par exemple, des pages HTML générées avec Thymeleaf).
- **`@Autowired`** : Cette annotation permet à Spring d'injecter automatiquement une dépendance. Dans ce cas, elle est utilisée pour injecter une chaîne de caractères `title`, probablement définie dans un fichier de configuration `.properties`. Cela illustre le mécanisme d'injection de dépendances basé sur les types ou les noms des beans.
- **`@GetMapping("authors-list")`** : Spécifie que la méthode suivante est liée à une requête HTTP GET envoyée à l'URL `/authors-list`. Cette annotation est une version abrégée de `@RequestMapping(method = RequestMethod.GET)`.

Elle est utilisée ici pour retourner une vue HTML contenant la liste des auteurs.

— **`@GetMapping("/biblio/authors")`** : Même rôle que la précédente, mais cette fois pour une route différente. Elle déclenche l'exécution de la méthode qui retourne la liste des auteurs en format brut (JSON).

— **`@ResponseBody`** : Indique que la valeur retournée par la méthode n'est pas une vue mais une donnée brute (par défaut au format JSON). Elle est utilisée ici pour exposer une méthode en tant que service REST. Cela permet à des clients externes (comme des applications front-end ou mobiles) de consommer directement les données.

Résumé des annotations :

Annotation	Rôle
`@Controller`	Définit la classe comme contrôleur Spring MVC pour retourner des vues.
`@Autowired`	Permet l'injection automatique d'une dépendance ou d'une propriété.
`@GetMapping("...")`	Associe une méthode à une requête HTTP GET sur une URL donnée.
`@ResponseBody`	Retourne directement la réponse (généralement en JSON) au client HTTP.

- application.properties :

Le fichier `application.properties`, situé dans le dossier `src/main/resources`, est utilisé pour la configuration externe de l'application Spring Boot.

Dans ce projet, il contient :

```
1 spring.application.name=p05-spring-boot-jsp-1
```

Listing 2 – Fichier application.properties

Cette propriété permet de donner un nom à l'application Spring Boot. Bien qu'elle ne soit pas utilisée directement dans le code, elle peut apparaître dans les logs de démarrage de Spring Boot, dans les tableaux de bord de monitoring, ou être utilisée pour identifier le service dans un système distribué (comme avec Eureka, Spring Cloud, etc.).

Ce fichier peut également être utilisé pour :

— définir le port d'écoute du serveur (`server.port=8081`),

— configurer une base de données (JDBC, JPA),

— activer/désactiver des fonctionnalités spécifiques,

— gérer les chemins des fichiers statiques ou templates.

Même avec une seule ligne, ce fichier montre l'importance de la configuration externe dans Spring Boot.

Lien logique entre les composants (flux de données)

Voici le chemin parcouru par les données :

```
Requête HTTP (navigateur) → Controller
                                ↓
                        BiblioService (métier)
                                ↓
                         AuthorDao (accès données)
                                ↓
                        Liste d'objets Author
                                ↓
                  Renvoyée à la vue JSP ou en JSON
```

Spring Boot automatise tout ce cycle grâce à ses annotations et à l'injection de dépendances.

3.4 p06-spring-boot

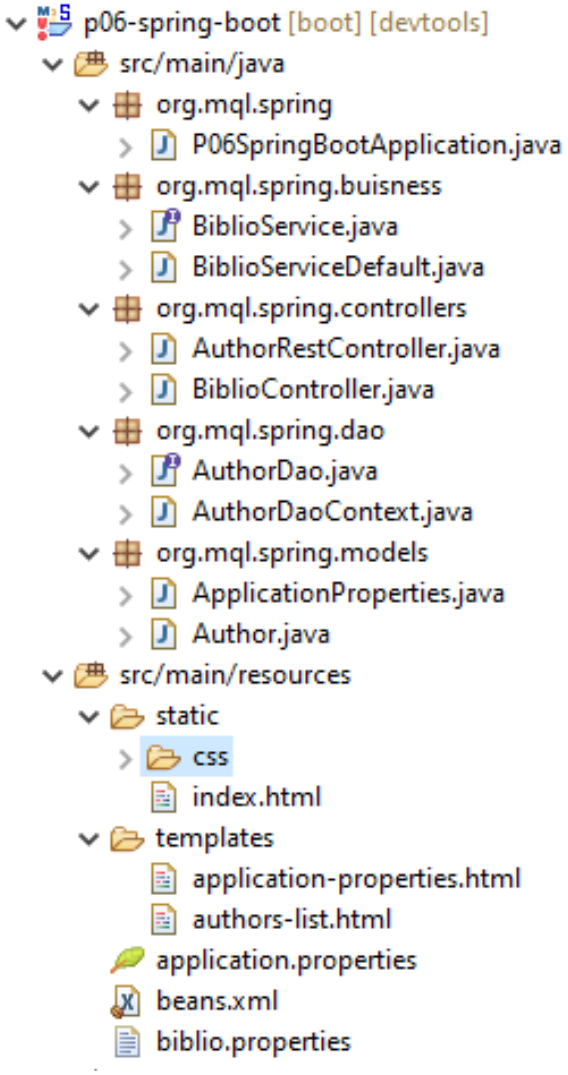

FIGURE 7 – architecture général

3.4.1 Architecture et organisation du code :

— `src/main/java/org.mql.spring` : paquet racine du code source Java.

— `business/` : Cette couche contient la logique fonctionnelle de l'application. Elle communique entre les contrôleurs et les DAO.

- BiblioService : interface déclarant la méthode métier getAllAuthors().

- BiblioServiceDefault : classe annotée @Service qui implémente l'interface. Elle utilise AuthorDao pour retourner les données à la couche supérieure

— `controllers/` : Les contrôleurs sont responsables de gérer les requêtes HTTP, que ce soit pour une API REST ou une application Web avec vues HTML.

- AuthorRestController : contrôleur REST (@RestController) exposant un endpoint /biblio/api/authors qui retourne la liste des auteurs en JSON.

- BiblioController : contrôleur Spring MVC (@Controller) qui affiche des vues Thymeleaf comme authors-list.html. Il injecte aussi les propriétés de configuration pour les afficher dans l'interface.

— `dao/` : Cette couche assure la gestion des données métier. Elle ne contient pas de logique métier, uniquement des opérations de lecture.

- AuthorDao : interface déclarant la méthode selectAll() pour récupérer tous les auteurs.

- AuthorDaoContext : classe implémentant l'interface AuthorDao et annotée @Repository. Elle simule une base de données avec une liste d'auteurs injectée.

— `models/` : Ce package regroupe les entités métiers ainsi que les classes de configuration de propriétés :

- ApplicationProperties : classe qui permet de lire les propriétés définies dans les fichiers .properties grâce aux annotations @Value

- Author : classe représentant un auteur avec des attributs id, name, yearBorn.

— `src/main/resources` : dossier contenant les ressources de l'application.

— `templates/` : contient les vues HTML utilisées par BiblioController, comme :

- authors-list.html : liste des auteurs affichée dans une table.

- application-properties.html :vue affichant les propriétés système et personnalisées.

— `application.properties` : configuration principale Spring Boot (nom de l'application, port, etc.).

— `biblio.properties` : fichier de propriétés personnalisées utilisé par la classe

— `beans.xml` : fichier de configuration Spring XML permettant de définir manuellement des beans (ex : Author).

— `pom.xml` : fichier de configuration Maven, qui déclare les dépendances (comme `spring-boot-starter-web`) et les plugins.

3.4.2 Objectif général du projet :

Ce projet a pour objectif principal de démontrer comment Spring permet de gérer efficacement des propriétés de configuration, qu'elles soient techniques, personnalisées, ou système, en les injectant dans des composants de l'application grâce à l'annotation @Value et aux fichiers de configuration .properties ou .xml.

Il s'agit d'un projet pédagogique visant à comprendre :

- Comment centraliser les informations de configuration d'une application Spring.

- Comment exploiter ces propriétés dans une classe métier.

- Comment afficher dynamiquement ces informations dans une interface utilisateur Web.

3.4.3 Objectif pédagogique structuré autour des types de propriétés :

a. Les propriétés de l'application (application.properties) : Situé dans le dossier resources/, ce fichier contient des propriétés techniques utilisées par le serveur Spring Boot :

- **spring.application.name :** nom de l'application.
- **server.port :** port HTTP de l'application.
- **server.servlet.context-path :** chemin de base de l'application.

Ces propriétés sont incontournables dans la configuration d'un projet Spring Boot, et sont injectées dans la classe **ApplicationProperties** pour être ensuite affichées dans l'interface Web.

b. Les propriétés personnalisées (biblio.properties) :

Un second fichier, **biblio.properties**, permet d'ajouter des valeurs définies par le développeur, comme :

- **mql.ds.name :** nom d'une source de données personnalisée.

Cela montre comment organiser et structurer des fichiers de propriétés dédiés à certains modules ou fonctionnalités de l'application.

c. Les propriétés système : Grâce à l'annotation **@Value**, le projet illustre aussi comment accéder à des propriétés système (Java ou OS) :

- **$user.name :** utilisateur courant du système.
- **$java.home :** répertoire d'installation de Java.
- **$java.runtime.name :** nom de la JVM.
- **$user.vendor :** fournisseur de la JVM.

Ces propriétés sont automatiquement extraites de l'environnement et injectées dans la classe **ApplicationProperties**, montrant la flexibilité de Spring dans la gestion de l'environnement d'exécution.

3.4.4 Mise en œuvre technique :

- Pour manipuler ces différentes propriétés, une classe dédiée a été créée :

```
@Component
@PropertySource({"classpath:biblio.properties"})
public class ApplicationProperties {
    @Value("${spring.application.name}")
    private String applicationName;

    @Value("${mql.ds.name}")
    private String sourceName;

    @Value("${user.name}")
    private String userName;

    // etc.
}
```

Grâce à l'annotation @Component, cette classe peut être injectée dans n'importe quel contrôleur Spring via @Autowired, ce qui permet d'accéder facilement aux propriétés depuis les classes d'action.

- Affichage dynamique dans une interface web :

Le contrôleur BiblioController est chargé d'extraire les propriétés injectées et de les envoyer à une vue Thymeleaf :

```
@GetMapping("props")
public String getProperties(Model model) {
    model.addAttribute("props", props); // props est une instance de
    ApplicationProperties
    return "application-properties";
}
```

- La vue application-properties.html affiche alors ces informations dans une table HTML structurée :

```
<td th:text="${props.applicationName}"></td>
<td th:text="${props.port}"></td>
<td th:text="${props.userName}"></td>
<!-- etc. -->
```

Ce projet constitue une mise en pratique complète de la gestion des propriétés avec Spring, en combinant :

- Injection de dépendances.
- Utilisation des annotations @Value, @PropertySource, @Component.
- Lecture des fichiers .properties.
- Intégration dans une interface utilisateur via Spring MVC + Thymeleaf.

Il met en évidence **l'un des grands avantages de Spring :** la possibilité de séparer la configuration de l'implémentation, tout en gardant une architecture claire, souple et maintenable.

3.5 p07-spring-boot-jsp

Le projet est une application Spring Boot qui gère une bibliothèque numérique. Il permet la gestion des auteurs, documents, éditeurs (publishers) via une architecture REST et MVC.

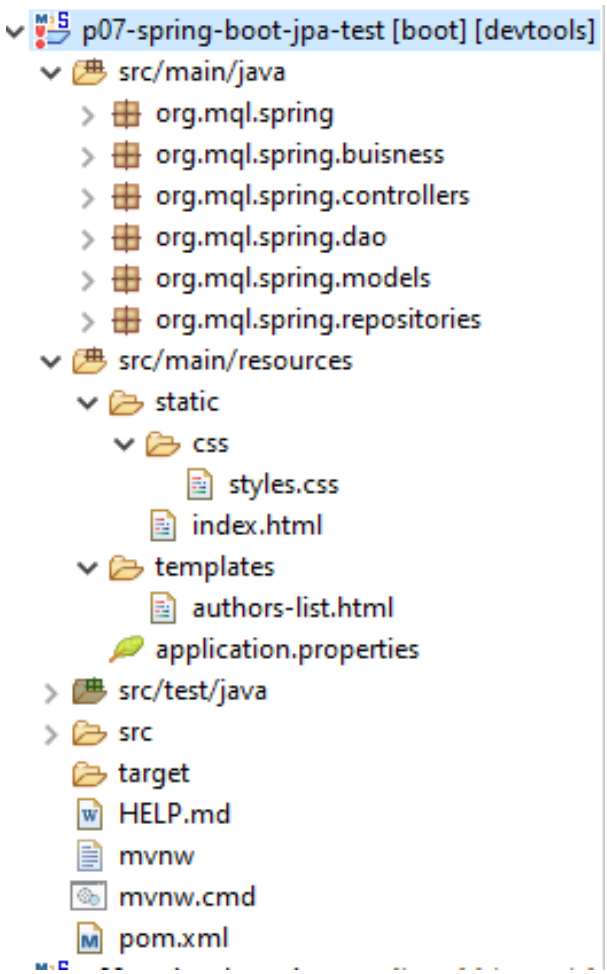

FIGURE 8 – architecture général

3.5.1 Architecture et organisation du code :

— `src/main/java/org.mql.spring` : paquet racine du code source Java.

- `business/` : Interface BiblioService et son implémentation BiblioServiceJpa qui encapsulent la logique métier, font le lien entre les repositories et les contrôleurs.
- `controllers/` : REST controllers exposant des API HTTP pour accéder aux données via JSON, et un contrôleur MVC pour la vue côté serveur.
- `dao/` : contient l'interface `AuthorDao` et la classe `AuthorDaoContext` annotée `@Repository` pour simuler l'accès aux données.
- `models/` : Entités JPA qui représentent les tables de la base de données : Author, Document, Publisher. Utilisation des annotations JPA (@Entity, @Table,

@Id, @ManyToOne, @ManyToMany), avec gestion des relations entre entités (ex : Document lié à plusieurs Author via table de jointure).

— `repositories/` : Interfaces qui étendent JpaRepository pour accéder à la base de données via JPA/Hibernate. Utilisation de méthodes dérivées (findByName, findByYearPublishedBetween) et requêtes JPQL personnalisées (@Query).

— **`src/main/resources`** : dossier contenant les ressources de l'application.

— **`src/main/resources/static`** :Ce dossier contient les ressources statiques accessibles directement (CSS, JS, images, fichiers HTML statiques). Par exemple, css/styles.css est un fichier CSS pour le style des pages.

— `application.properties` : fichier de configuration Spring. Il contient ici `spring.application`

— `static/index.html` : page statique accessible via le navigateur à `/index.html`.

— `templates/` : Contient les templates côté serveur, en général des fichiers HTML Thymeleaf ou JSP, utilisés pour générer des vues dynamiques. Ici, authors-list.html est un template probablement utilisé pour afficher la liste des auteurs.

— **`src/test/java`** : contient les classes de test (vide pour l'instant).

— **`target/`** : dossier généré automatiquement à la compilation contenant le fichier `.jar`, les classes compilées et les rapports de test.

— **`pom.xml`** : fichier de configuration Maven, qui déclare les dépendances (comme `spring-boot-starter-web`) et les plugins.

— **`mvnw, mvnw.cmd, .mvn/`** : fichiers pour utiliser Maven Wrapper, permettant d'exécuter Maven sans l'installer globalement.

— **`HELP.md`** : fichier d'aide généré automatiquement à la création du projet (optionnel).

3.5.2 Analyse détaillée du package :

- org.mql.spring.buisness :

Classe principale : BiblioServiceJpa

- Annotée **@Service**, cette classe est un composant Spring gérant la logique métier.

- Utilise l'injection automatique **(@Autowired)** pour accéder aux **AuthorRepository** et **DocumentRepository**.

- Implémente l'interface **BiblioService** (non fournie, mais supposée déclarer les méthodes exposées).

- Méthodes implémentées :

Méthode	Fonctionnalité	Remarques
getAllAuthors()	Récupère tous les auteurs	`authorRepository.findAll()`
getAuthorById(int id)	Récupère un auteur par son ID	Utilise `Optional` pour éviter les nulls
getAuthorsByName(String name)	Récupère auteurs par nom	Appelle `authorRepository.findByName(.)`
getAuthorsByKeyword(String keyword)	Récupère auteurs dont le nom commence par `keyword`	Utilise requête JPQL personnalisée
getAllDocuments()	Récupère tous les documents	Appelle `documentRepository.findAll()`
getDocumentByIsbn(String isbn)	(Non implémenté)	À implémenter
getDocumentsByKeyword(String keyword)	(Non implémenté)	À implémenter
getDocumentsByYearInterval(int min, int max)	Récupère documents publiés entre 2 années	`documentRepository .findByYearPublishedBetween(min, max)`

Table 3 – Résumé des méthodes du service bibliographique

- Contrôleurs REST :

- AuthorRestController :

- Expose l'API /api/authors
- Méthodes REST :

Endpoint	Description	Retour
GET /api/authors	Liste de tous les auteurs	`List<Author>`
GET /api/authors/id	Auteur par ID	Author avec gestion 404 si non trouvé
GET /api/authors ?keyword=xxx	Liste des auteurs dont le nom commence par keyword	AList<Author>

Table 4 – Méthodes REST

- DocumentRestController :

- Expose l'API /api/documents
- Méthodes REST :

Endpoint	Description	Retour
GET /api/documents	Liste tous les documents	`List<Document>`
GET /api/documents ?min=xx&max=yy	Documents publiés entre deux années	List<Document>

Table 5 – Méthodes REST

- BiblioController :

- Expose la page web /biblio/authors-list

- Récupère la liste des auteurs depuis le service et la place dans le modèle pour affichage via une vue JSP ou Thymeleaf (non fournie).

- Injecte aussi une variable title (probablement pour affichage dans la page).

- org.mql.spring.repositories :

- AuthorRepository :étend JpaRepository sur Author

- Méthode dérivée findByName pour rechercher auteurs par nom exact.

- Requête JPQL personnalisée findByKeyword pour recherche partielle sur le nom (avec LIKE).

- DocumentRepository :étend JpaRepository sur Document.

- Méthode dérivée findByName pour rechercher auteurs par nom exact.

- Requête JPQL personnalisée findByKeyword pour recherche partielle sur le nom (avec LIKE).

- Modèles de données (Entities) :

- Author : id, nom, année de naissance, mappé à la table Authors.

- Document : isbn, titre, année, publisher, liste d'auteurs (relation @ManyToMany avec table Document_Author).

- Publisher : id, nom, société, table publishers.

3.6 p08-spring-boot-jpa

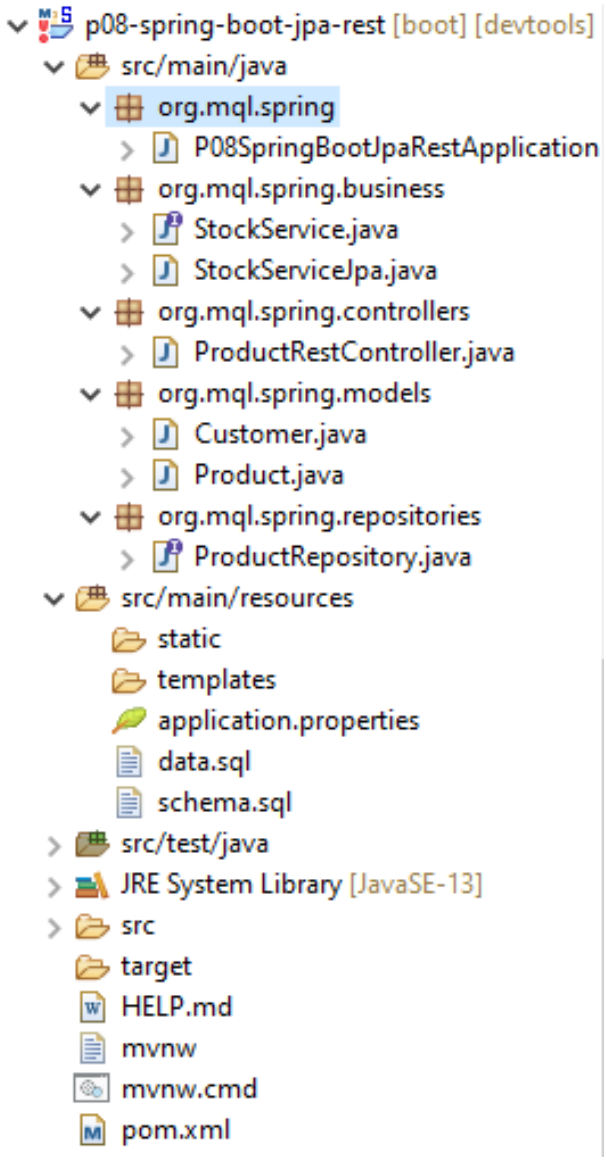

FIGURE 9 – architecture général

3.6.1 Architecture et organisation du code :

— `src/main/java/org.mql.spring` : paquet racine du code source Java.

— `business/` : Cette couche contient la logique fonctionnelle de l'application. Elle agit comme intermédiaire entre les contrôleurs (couche web) et les repositories (couche données).

- StockService : Interface définissant les opérations métiers sur les produits (liste, ajout, recherche par ID).

- StockServiceJpa : Implémentation concrète de l'interface `StockService` utilisant Spring Data JPA pour manipuler les produits via le repository.

- `controllers/` : Les contrôleurs sont responsables de gérer les requêtes HTTP reçues de l'extérieur. Ils font appel aux services métiers pour exécuter la logique souhaitée.

  - ProductRestController : Contrôleur REST exposant une API permettant d'afficher la liste des produits et d'ajouter de nouveaux produits au stock. Les données sont échangées au format JSON.

- `models/` : Ce package regroupe les entités métiers de l'application, qui sont directement mappées sur les tables de la base de données via JPA.

  - Product : Entité représentant un produit avec les attributs `id`, `name`, `price` et `quantity`.

  - Customer : Entité représentant un client avec les attributs `id`, `name`, et `email`.

- `repositories/` : Cette couche représente l'accès aux données. Elle est basée sur l'interface `JpaRepository` fournie par Spring Data JPA.

  - ProductRepository : Interface héritant de `JpaRepository<Product, Integer>` permettant d'effectuer les opérations CRUD sur les entités `Product` sans implémentation explicite.

- **`src/main/resources`** : dossier contenant les fichiers de configuration et les ressources SQL.
  - `application.properties` : Contient les paramètres de configuration de l'application comme le nom, le port, le contexte, la configuration de la base de données et les paramètres JPA.
  - `schema.sql` : Fichier SQL permettant la création de la table `Product` si elle n'existe pas.
  - `data.sql` : (non utilisé ici) peut servir à insérer des données initiales dans la base lors du démarrage.
- **`pom.xml`** : Fichier de configuration Maven. Il déclare les dépendances nécessaires au projet, telles que `spring-boot-starter-web`, `spring-boot-starter-data-jpa`, `mysql-connector-java`, etc. Il contient également la configuration du plugin `spring-boot-maven-p` pour l'exécution du projet.

Objectifs et apports pédagogiques du projet

Ce projet intitulé **`p08-spring-boot-jpa-rest`** a pour objectif de mettre en pratique la création d'un service web REST complet à l'aide du framework **Spring Boot**, avec persistance des données via **Spring Data JPA** et stockage dans une base de données **MySQL**.

Les objectifs principaux sont les suivants :

- Concevoir une architecture en couches (contrôleur, service, repository, modèle).
- Exposer une API REST permettant d'accéder aux produits (GET) et d'en ajouter (POST).
- Gérer la persistance des objets Java (produits) dans une base MySQL.

— Utiliser les fichiers de configuration Spring Boot pour paramétrer l'application.

Ce projet introduit plusieurs **notions techniques avancées**, parmi lesquelles :

— **Spring Data JPA** : gestion automatique des opérations CRUD via `JpaRepository`, sans implémentation manuelle.
— **Annotations JPA** : utilisation de `@Entity`, `@Id`, etc., pour mapper les classes Java sur les tables de la base de données.
— **API REST** : création de services web RESTful via `@RestController`, `@GetMapping` et `@PostMapping`.
— **Sérialisation JSON** : conversion automatique des objets via `@RequestBody` pour les requêtes AJAX.
— **Configuration externe** : gestion centralisée de la configuration (base de données, contexte, port, etc.) via `application.properties`.
— **Initialisation de base de données** : création de tables via `schema.sql`, avec possibilité d'insérer des données initiales via `data.sql`.

Ce projet constitue une base solide pour la création d'applications d'entreprise reposant sur des services REST et une base relationnelle.

Focus fonctionnel : Intégration de la base de données

Ce projet introduit une nouvelle dimension essentielle dans le développement d'applications web : la **connexion à une base de données relationnelle** et la **persistance des entités métiers**. Grâce à **Spring Data JPA**, le projet permet de manipuler des objets Java (produits) tout en les enregistrant automatiquement dans une base de données `MySQL`.

Pour mettre en place cette fonctionnalité, plusieurs éléments techniques ont été ajoutés ou configurés :

1. Dépendances Maven Le fichier `pom.xml` a été enrichi avec les dépendances suivantes :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.30</version>
</dependency>
```

— `spring-boot-starter-data-jpa` : fournit l'infrastructure JPA
— `mysql-connector-java` : pilote JDBC permet la connexion entre Spring et la base MySQL.

2. Fichier `application.properties` La configuration de la connexion à la base de données se fait via ce fichier :

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/Stock
spring.datasource.username=root
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=update
spring.sql.init.mode=always
#spring.sql.init.data-locations=classpath
```

L'instruction `ddl-auto=update` permet à Hibernate de créer ou mettre à jour automatiquement les tables à partir des entités Java.

3. Création de schéma SQL Le fichier `schema.sql` a été ajouté pour initialiser la base de données avec la table

```
DROP TABLE IF EXISTS Product;

CREATE TABLE Product(
  id INT,
  name VARCHAR(50),
  price DOUBLE,
  quantity DOUBLE,
  PRIMARY KEY (id)
)
```

`Product`. Cette table correspond à l'entité `Product` annotée avec `@Entity`.

3. Création de data.sq Le fichier `schema.sql` a été ajouté pour initialiser la base de données avec la table

```
@Entity
public class Product {
  @Id
  private int id;
  private String name;
  private double price;
  private double quantity;
  // getters, setters, constructeur
}
```

Annotations :

@Entity → déclare que cette classe est persistable (liée à une table).

@Id → indique la clé primaire.

5. Interface `ProductRepository` Cette interface étend `JpaRepository<Product, Integer>`, permettant d'effectuer automatiquement les opérations CRUD sans code supplémentaire.

```java
public interface ProductRepository extends JpaRepository<Product, Integer>
    {}
```

Rôle :

Fournit automatiquement toutes les méthodes CRUD (findAll(), save(), etc.) sans implémentation manuelle.

Gère les interactions directes avec la base

6. Service métier : StockService et StockServiceJpa Interface StockService :

```java
public interface StockService {
  List<Product> getAllProducts();
  Optional<Product> getProductById(int id);
  boolean addProduct(Product product);
}
```

Classe StockServiceJpa :

```java
@Service
public class StockServiceJpa implements StockService {
  @Autowired
  private ProductRepository productRepository;

  @Override
  public List<Product> getAllProducts() {
    return productRepository.findAll();
  }

  @Override
  public boolean addProduct(Product product) {
    productRepository.save(product);
    return true;
  }
}
```

`Annotations` :

— @Service : rend la classe injectable dans d'autres composants Spring.

— @Autowired : injecte automatiquement une instance de ProductRepository.

Le contrôleur REST ProductRestController

```java
@RestController
@RequestMapping("/api/products")
@CrossOrigin("*")
public class ProductRestController {

  @Autowired
  private StockService service;

  @GetMapping
  public List<Product> products() {
    return service.getAllProducts();
```

```
  }

  @PostMapping
  public ResponseEntity<Product> add(@RequestBody Product p) {
    service.addProduct(p);
    return ResponseEntity.ok(p);
  }
}

```

`Annotations` :

- @RestController : indique qu'il s'agit d'un contrôleur REST (retourne du JSON).
- @RequestMapping : base des URI.
- @CrossOrigin("*") : autorise les appels depuis n'importe quelle origine.
- @GetMapping et @PostMapping : mappent les requêtes GET et POST.
- @RequestBody : convertit automatiquement le JSON en objet Java (désérialisation).

Étape	**Composant**	**Fonction**
1	`ProductRestController`	Reçoit une requête HTTP
2	`StockService`	Encapsule la logique métier
3	`ProductRepository`	Interagit avec la base de données
4	`Product`	Entité mappée à une table via JPA
5	`application.properties` + `pom.xml`	Configurent la connexion JDBC, JPA, et le pilote MySQL

TABLE 6 – Étapes du processus de persistance des données

Grâce à cette configuration, le projet met l'accent sur la maîtrise de la couche **données** dans une architecture Spring complète, en abordant la configuration, le mapping objet-relationnel et la manipulation via les services REST.

4 Comparaison entre JEE native, Spring, et Spring Boot :

Critère	**JEE native**	**Spring (XML / Annotations)**	**Spring Boot**
Configuration	Lourde, basée sur XML (web.xml, ejb-jar.xml)	XML ou Java Annotations (@Bean, @Configuration)	Auto-configuration, très simplifiée
Démarrage du projet	Nécessite un serveur d'application (ex : GlassFish)	Nécessite Tomcat/-Jetty externe ou intégré	Serveur embarqué intégré (Tomcat, Jetty)
Injection de dépendance	JNDI, @EJB (verbeux)	IoC avec ApplicationContext, @Autowired	Automatique avec @SpringBootApplication
Productivité	Faible (complexité de configuration)	Moyenne (modularité mais config manuelle)	Élevée (starters, devtools, build rapide)
Architecture	Couplée, difficilement testable	Architecture en couches bien définie	Architecture claire, microservices-ready
Courbe d'apprentissage	Élevée	Modérée	Faible à modérée
Déploiement	EAR/WAR via serveur distant	WAR local/distant	JAR exécutable ("java -jar")
Utilisation en entreprise	Présente dans projets legacy	Courante dans projets stables	Dominante dans projets modernes (cloud, REST)

Table 7 – Comparaison entre JEE native, Spring, et Spring Boot

5 Conclusion

L'évolution de Java EE vers Spring et ensuite vers Spring Boot reflète un besoin croissant de simplicité, de modularité et de rapidité dans le développement des applications d'entreprise. Spring a marqué une avancée majeure avec son modèle d'injection de dépendances et son système de configuration souple. Spring Boot, quant à lui, a poussé cette logique plus loin en proposant une approche "convention over configuration", permettant aux développeurs de se concentrer sur la logique métier plutôt que sur les aspects techniques de configuration. Aujourd'hui, Spring Boot s'impose comme le standard pour développer rapidement des applications web modernes, robustes, et évolutives.

Références

[1] https://docs.spring.io/spring-framework/docs/current/reference/html/.

[2] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/.

[3] https://jakarta.ee/specifications/persistence/.