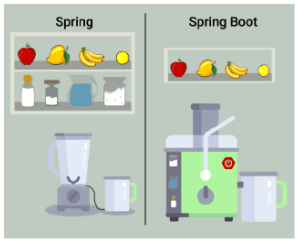
TP 1 : Création d'une application Web de Gestion de Produits avec Spring Boot

Objectif du TP

L'objectif final de ce TP est de développer une application Spring web MVC comme étant un projet Spring Boot permettant la gestion (CRUD) d'une BD composée de 2 tables reliées entre elles produit et catégorie.

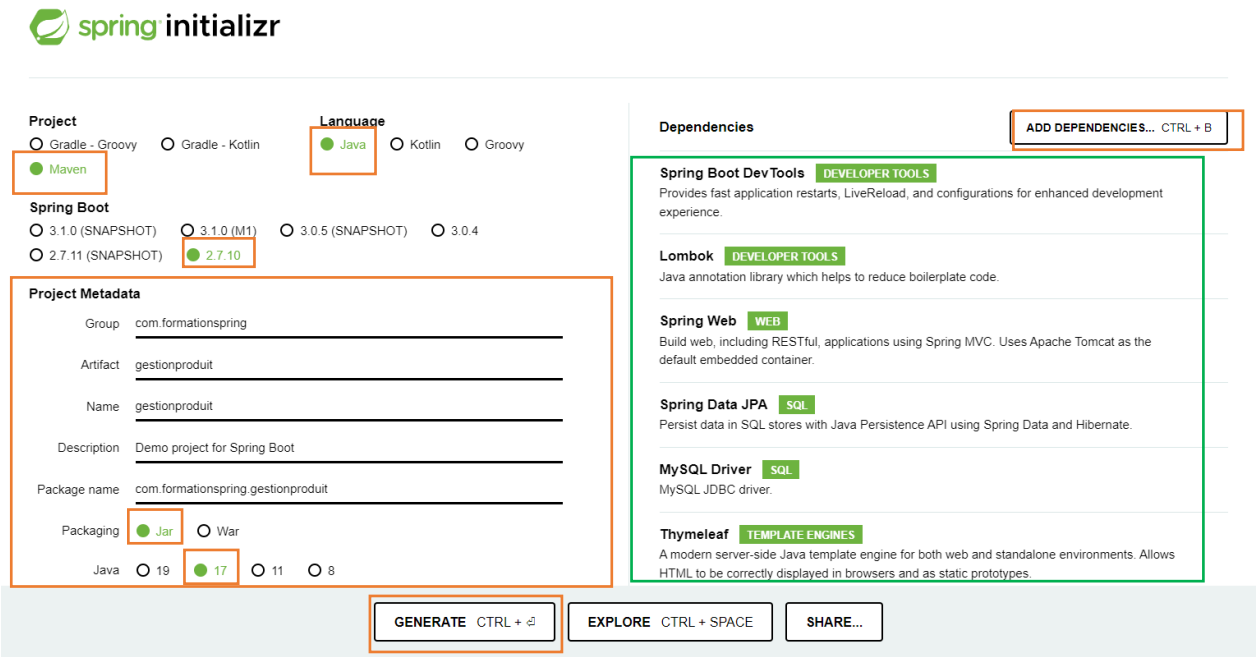
Spring vs Spring Boot



Spring	Spring Boot
• **Le** framework Spring est l'un des frameworks les plus populaires pour le développement des applications en Java. • Il vise à simplifier le développement Java EE qui rend les développeurs plus productifs. • Les principales caractéristiques de Spring Framework sont **l'inversion de control et** l' **injection de dépendances** . • Cela aide à simplifier les choses en nous permettant de développer **des** applications **faiblement couplées** . • Pour tester le projet Spring, nous devons configurer le serveur explicitement. • Les développeurs définissent manuellement les dépendances pour le projet Spring dans **pom.xml** .	• Spring Boot est une extension du framework Spring, • **Spring Boot Framework** est largement utilisé pour développer **des API REST** • Il vise à raccourcir la longueur du code et à fournir le moyen le plus simple de développer **des applications Web** . • La principale caractéristique de Spring Boot est la **configuration automatique** . Il configure automatiquement les classes en fonction des besoins. • Il permet de créer une application **autonome** avec moins de configuration • Spring Boot propose **des serveurs embarqués** tels que **Jetty** et **Tomcat** , etc. • Spring Boot est livré avec le concept de **démarreur** dans le fichier pom.xml qui prend en charge en interne le téléchargement des **JAR de** dépendances en fonction des exigences de Spring Boot.

Création d'un projet Spring Boot

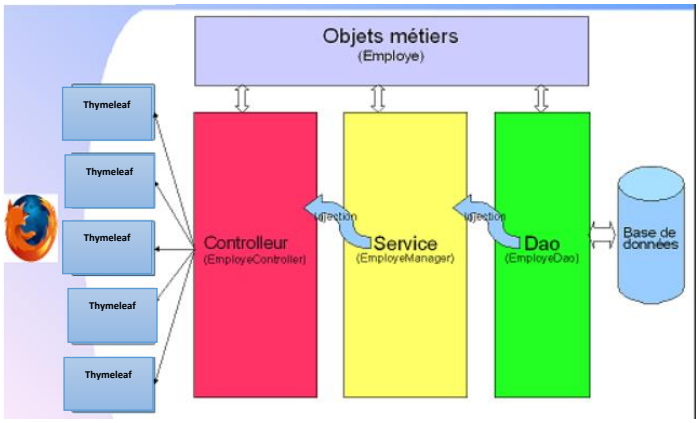
- Taper l'adresse suivante =>https://start.spring.io/



- Télécharger(Generate) et importer le projet « maven » dans Eclipse

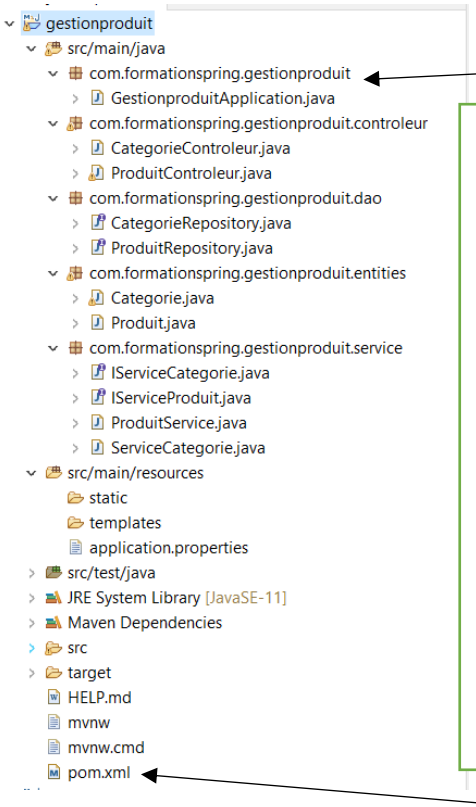
Maven va télécharger les dépendances depuis le dépôt central vers le dépôt local (le dossier .m2 situé sous C:\users\votre_session). Pour la première création d'un projet, le téléchargement peut prendre quelques minutes selon le débit de la connexion.

Bonne pratique : Architecture d'une application Spring Boot MVC



Structure du projet

La structure finale du projet correspondant à notre application devrait être :



Le package principale du projet `com.formationspring.gestionproduit` contient la classe exécutable `GestionProduitApplication.java`; qui le point de démarrage de l'application Spring Boot.

En respectant l'architecture d'une application Spring Boot, Vous devez créer respectivement les sous packages suivants :

- ***com.formationspring.gestionProduit.entities*** contenant la définition des entités Produit et Categorie.
- ***com.formationspring.gestionproduit.dao*** contenant 2 interfaces ProduitRepository et CategorieRepository héritant chacune de l'interface générique JpaRepository<T,ID>.
- Les paramètres d'accès à la BD sont définis dans le fichier ***application.properties*** sous *src/main/resources.*

→Ces 3 composants représentent la couche persistance dans l'architecture de l'application.

- ***com.formationspring.gestionProduit.service*** contenant 2 interfaces qui contiennent chacune la signature des méthodes métier sur les entités Produit et Categorie et 2 classes (annotées par @Service) implémentant chacune les méthodes métier déclarées dans chacune des interfaces. Le contenu de ce package représente la couche métier (service) de l'application.
- ***com. formationspring.gestionProduit.controller*** contenant 2 classes dont chacune joue le rôle d'un contrôleur web (annoté par @Controller) et contiendra les actions traitant les requêtes HTTP. Chaque action retournera soit le nom de la vue, soit une redirection vers un URL d'une autre action. Le contenu de ce package représente la couche web de l'application.

Fichier de dépendances Maven

Etape 1 : Définition des entités

Lombok est une dépendance qui a pour but de réduire le code des classes entités en remplaçant la définition des setters, getters et constructeurs par des annotations.

- L'annotation @Data permet de remplacer tous les getters et setters des attributs.
- L'annotation @NoArgsConstructor permet de remplacer le constructeur par défaut
- L'annotation @AllArgsConstructor permet de remplacer le constructeur avec paramètres
- L'annotation @Builder de Lombok est utilisée pour implémenter facilement le design pattern **Builder** dans une classe Java. Ce pattern permet de construire des objets de manière plus flexible, en particulier lorsqu'une classe a beaucoup de champs ou lorsque des combinaisons d'attributs sont optionnelles.

Avant de commencer le développement et pour bien utiliser Lombok, il faut indiquer dans quel IDE on va l'utiliser. Pour cela, dans l'explorateur de Windows ouvrez le dossier :

C:\Users*votre_session*\.m2\repository\org\projectlombok\lombok\1.18.22, double-cliquer sur le fichier lombok-1.18.22.jar, la fenêtre suivante apparait, choisir l'emplacement de votre IDE et valider. Enfin Redémarrer Eclipse.

- Créer deux entités « Produit » et « Categorie ». L'association entre les deux classes est une composition :


```
import javax.persistence.Entity;
@Data
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Entity
public class Produit {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private double prix;
    private int quantite;
    private String photo;
    @ManyToOne
    private Categorie categorie;
}
```

à La classe Produit correspondra une table produit

id AUTO-INCREMENT dans la table produit

id sera la clé primaire dans la table produit

L'annotation @ManyToOne définit une relation n:1 entre deux entités. L'annotation @ManyToOne implique que la table Produit contient une colonne qui est une clé étrangère contenant la clé d'une Catégorie.

Chaque Produit est caractérisé par sa catégorie

C'est le nom qui est associé au paramètre mappedBy

```
import java.util.ArrayList;
@Data
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Entity
public class Categorie {

    @Id @GeneratedValue(strategy =GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    @OneToMany(mappedBy = "categorie",cascade = CascadeType.ALL)
    private List<Produit>liste=new ArrayList<Produit>();

    }
```

L'annotation @OneToMany définit une relation 1:n entre deux entités. Cette annotation ne peut être utilisée qu'avec une collection d'éléments puisqu'elle implique qu'il peut y avoir plusieurs entités associées.

On utilise le paramètre *mappedBy* pour indiquer l'entité esclave. *mappedBy* reçoit le nom de l'objet de type Categorie déclaré dans la classe Produit.

Chaque Categorie a une liste (Collection) de Produit

La signification de ***CascadeType.ALL*** est que la persistance propagera (cascade) toutes les opérations EntityManager (PERSIST, REMOVE, REFRESH, MERGE, DETACH) aux entités associées.

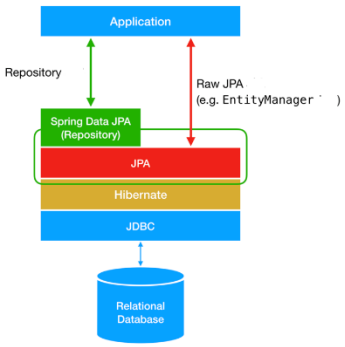
Etape 2 : Définir les paramètres d'accès à la BD

- Ajouter dans le fichier *application.properties* les paramètres suivants :

```
#Server configuration
#server.port=8081
### DATABASE ###
#spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/catalogue2022?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
### JPA / HIBERNATE ###
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

- Exécuter votre application pour générer les tables dans la BD.

Etape 3 : Création des Repository



- Créer deux interfaces nommées respectivement CategorieRepository, ProduitRepository comme suit :

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface CategorieRepository extends JpaRepository<Categorie, Integer> {

}
```

```
public interface ProduitRepository extends JpaRepository<Produit, Integer> {


}
```

Ces deux interfaces héritent toutes les méthodes CRUD de ***JpaRepository***.

- Ajouter la signature d'une méthode qui recherche les produits par leurs noms. Utiliser des requêtes personnalisées.

```
public interface ProduitRepository extends JpaRepository<Produit,Integer> {

    public List<Produit> findByNomContains(String mc);

    @Query("select p from Produit p where p.categorie.id=:x")
    public List<Produit> getPructsByCat(@Param("x") Integer idc);
}
```

Derived Queries

Requête JPQL

Etape 4 : Création de la couche service (métier)

- Ajouter deux interfaces IgestionProduit, IgestionCategorie contenant les méthodes suivantes :

```java
package com.formationspring.gestionproduit.service;

import ...
public interface IServiceProduit {
    public void saveProduct(Produit p);

    public List<Produit> getAllProducts();

    public Produit getProduct(Integer id);

    public void deleteProduct(Integer id);

    public List<Produit> getProductsBMC(String mc);

    public List<Produit> getProductsBCategorie(Integer idCat);


}
```

```java
package com.formationspring.gestionproduit.service;

import ...

public interface IserviceCategorie {
    public void addCategorie(Categorie c);

    public List<Categorie> getAllCategorie();
}
```

- Dans le même package, créer la classe ServiceProduit qui implémente l'interface IServiceProduit et la classe ServiceCategorie qui implémente l'interface IServiceCategorie

L'injection de dépendance consiste à éviter une dépendance directe entre deux classes, et définissant **dynamiquement** la dépendance plutôt que statiquement.

```java
import ...
@Service
@AllArgsConstructor
public class ServiceProduit implements IServiceProduit {
    private ProduitRepository pr;
    @Override
    public void saveProduct(Produit p) { pr.save(p); }

    @Override
    public List<Produit> getAllProducts() { return pr.findAll(); }

    @Override
    public Produit getProduct(Integer id) { return pr.findById(id).orElse( other: null); }

    @Override
    public void deleteProduct(Integer id) { pr.deleteById(id); }

    @Override
    public List<Produit> getProductsBMC(String mc) { return pr.findByNomContains(mc); }

    @Override
    public List<Produit> getProductsBCategorie(Integer idCat) { return pr.getPructsByCat(idCat); }


}
```

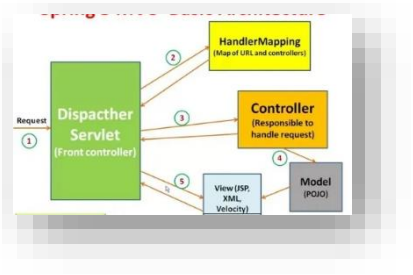
```java
package com.formationspring.gestionproduit.service;
import ...

@Service
@AllArgsConstructor
public class ServiceCategorie implements IserviceCategorie{
    private CategorieRepository categorieRepository;
    @Override
    public void addCategorie(Categorie c) {categorieRepository.save(c); }

    @Override
    public List<Categorie> getAllCategorie() { return categorieRepository.findAll(); }
}
```

Etape 5 : Définition de la couche web

Dans cette partie nous intéressons à la création d'un contrôleur pour manipuler les produits

Un contrôleur est une classe java annotée par Controller (ou bien RestController). C'est un composant du modèle MVC.

Spring MVC : Architecture de base



- Dans le sous-package « controleur », créer la classe suivante :

Ici L'annotation @GetMapping(« /home ») signifie que les requêtes http de type Get à l'url /home exécuteront le code de la méthode getproducts ()

Pour tester,l'invocation de cette méthode, il faut taper l'url suivante :
localhot :8080/home

`@RequestParam(value = "mc") String mc` : permet de récupérer la valeur du paramètre de la requête HTTP est de l'affecter au paramètre mc de la méthode.

Nous pouvons utiliser l'annotation @PathVariable pour extraire la valeur de l'URL.

```
@Controller
@AllArgsConstructor
public class ProduitControleur {
    private IServiceProduit serviceProduit;
    private IserviceCategorie serviceCategorie;
    @GetMapping(path = {"/home"})
    public String getProducts(Model m,@RequestParam(name="mc",defaultValue = "")String mc)
    {  // m.addAttribute("data",serviceProduit.getAllProducts());
        m.addAttribute( attributeName: "data",serviceProduit.getProductsBMC(mc));
        m.addAttribute( attributeName: "mc",mc);
        return "home";  }

    @GetMapping("/delete/{id}")
    public String deleteProducts(@PathVariable Integer id)
    {   serviceProduit.deleteProduct(id);
        return "redirect:/home";   }

    @GetMapping("/edit/{id}")
    public String editProduct(@PathVariable Integer id,Model m)
    {   m.addAttribute( attributeName: "produit",serviceProduit.getProduct(id));
        m.addAttribute( attributeName: "categories",serviceCategorie.getAllCategorie());
        return "ajouter";   }
    @PostMapping("/save")
    public String saveProduct(@ModelAttribute Produit p, Model m) {
        serviceProduit.saveProduct(p);
        return "redirect:/home";
    }
   @GetMapping("/formProduit")
    public String redirection(Model m)
   {   m.addAttribute( attributeName: "categories",serviceCategorie.getAllCategorie());
       m.addAttribute( attributeName: "produit",new Produit());
       return "ajouter"; }
}
```

Dans la déclaration de la méthode, on injecte l'interface Model qui nous permettra d'envoyer des attributs à la vue. Dans notre cas, la vue à retourner représente la page html « home.html »

Etape 6 : Création des vues

Le moteur de Template **Thymeleaf** est utilisé pour générer des vues HTML dans une application web MVC. Il se base sur un ensemble d'attributs ajoutés dans les balises HTML de la vue dont le moteur de Template va les interpréter et les traduire en code HTML.

Les vues Thymeleaf ont l'extension .html et doivent être placées dans le dossier ***templates*** sous src/main/resources.

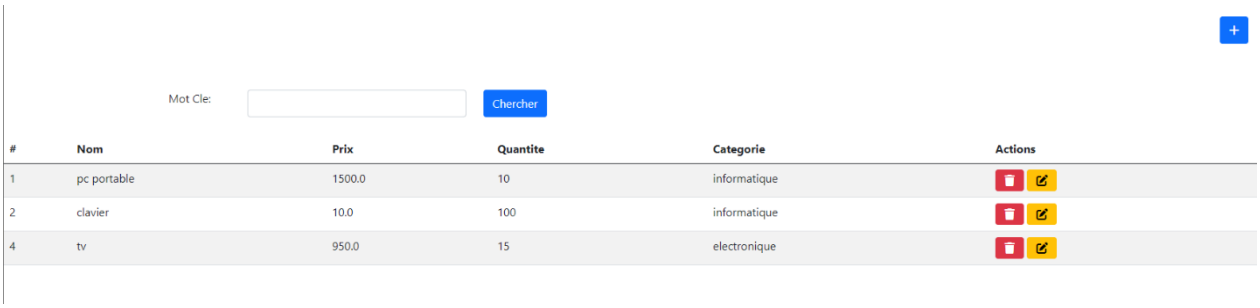
Pour utiliser Thymeleaf dans une vue, on commence par déclarer l'utilisation de namespace Thymeleaf dans la balise HTML de la page :

<**html** xmlns:th="http://www.thymeleaf.org">

Ensuite, selon notre besoin on utilise les attributs spécifiques.

```
<body>
<br>
<form method=get th:action="@{/home}">
    <div class=container>
        <div class="row mt-4 pl-2">
            <div class="col-md-1">
                <label for="mc" class="form-label">Mot Cle:</label>
            </div>
            <div class="col-md-3">
                <input type="text" id="mc" class="form-control" name=mc th:value="${mc}">
            </div>
            <div class="col-md-3">
                <button type="submit" class="btn btn-primary">Chercher</button>
            </div>
        </div>
    </div>
</form>
<br>
<table class="table table-striped">
    <thead>
    <tr>
        <th>#</th>
        <th>Nom</th>
        <th>Prix</th>
        <th>Quantite</th>
        <th>Categorie</th>
        <th>Actions</th>
    </tr>
    </thead>
    <tbody>
    <tr th:each="p:${data}">
        <td th:text="${p.id}"></td>
        <td th:text="${p.nom}"></td>
        <td th:text="${p.prix}"></td>
        <td th:text="${p.quantite}"></td>
        <td th:text="${p.categorie.nom}"></td>

<td>
            <a th:href="@{/delete/{id}(id=${p.id})}" class="fa fa-trash btn btn-danger"></a>
            <a th:href="@{/edit/{id}(id=${p.id})}" class="fa fa-edit btn btn-warning"></a>
        </td>
    </tr>
    </tbody>
</table>
</body>
</html>
```

Home.html




```
<body>
<br>
<div class=container>
    <div class="card m-5 p-2">
        <div class="card-header"><h2>Ajouter Produit</h2></div>
        <div class="card-body">
          <form method="post"  th:action="@{/save}">
    <div class="form-group">
        <label for="nom">Nom:</label>
        <input type="text" name="nom" class="form-control" id="nom"  th:value="${produit.nom}">
    </div>
    <div class="form-group">
        <label for="prix">Prix:</label>
        <input type="number" name="prix" class="form-control" id="prix"  th:value="${produit.prix}">
    </div>
    <div class="form-group">
        <label for="quantite">Quantite:</label>
        <input type="number" name="quantite" class="form-control" id="quantite"  th:value="${produit.quantite}">
    </div>
    <div class="form-group">
        <label for="categorie">Categorie</label>
        <select class="form-control" id="categorie" name="categorie">
            <option selected="selected" hidden=""></option>
            <option th:each="c:${categories}" th:value="${c.id}" th:text="${c.nom}"
                    th:selected="${produit.categorie!=null && produit.categorie.id==c.id}"></option>
        </select>
    </div>
    <br>
   <input type="hidden" name="id" th:value="${produit.id}">
    <button type="submit" class="btn btn-primary">Submit</button>

          </form></div></div></div>
</body>
```

Ajouter.html