

SPRING

Séance 1 :

Framework Spring : faciliter et de rendre productif le développement d'applications.

Séance 2 : Maven :

Maven est un outil qui nous permet la construction des projets, automatiser les tâches, faire gestion des dépendances (point fort)

Version 3.0

Version 3.0.1 → cad on a ajouté une nouvelle fonctionnalité

Version 3.1 → correctif = patch

Version 4.0 → nouveau livrable

- ❖ JAR : java archive
- ❖ WAR : version web (web archive)
- ❖ Pom : mixte entre les deux (Project Object model)

L'une des forces de maven → utilise **(Convention over configuration)** :

Maven préfère la convention à la configuration

Cycle de vie d'un projet maven :

- Mvn compile : Créer les .class
- Mvn test : compile (crée .class) + test (lance les tests unitaires)
- Mvn install : compile + test + package (préparation livrable sous Target) + Install (livrable target → repo local) : installer l'application dans un repository local
- Mvn package : compile + test + package
- Mvn deploy : deploy (envoyer notre livrable .jar au répertoire distant)
- Mvn clean : Supprime le contenu du dossier Target
- Clean Install : supprimé l'existant au niveau de Target et puis on met le nouveau JAR

Un projet mavenisé : un projet qui ne contient pas le **POM**

Pour le projet web : il faut générer Deployment descriptor stub

Séance 3 Log4j (journalisation):

Log4j est un Api de journalisation très répandue, ça remplace le système.Out.println (manque de mise en forme, stocké dans le même fichier)

■ La bibliothèque log4j met 3 sortes de composants à disposition du programmeur :

✚ Les loggers : permette d'écrire les messages :


- ❖ Log4j gère des priorités, ou Level, pour permettre au logger de déterminer si le message sera envoyé dans le fichier de log (ou la console).
- ❖ Il existe six priorités qui possèdent un ordre hiérarchique croissant :
TRACE, DEBUG, INFO, WARN, ERROR, FATAL (faible -> fort)

✚ Les appenders : servent à sélectionner la destination des messages :

- ❖ Pour ajouter un appender à un logger, il suffit de le rajouter dans le fichier configuration log4j.xml:

```
<root>
  <level value="WARN" />
  <appender-ref ref="console" />
  <appender-ref ref="file" />
</root>
```

- ❖ Un logger peut posséder plusieurs appenders
- ❖ L'interface org.apache.log4j.Appender désigne un flux qui représente le fichier de log et se charge de l'envoi de message formaté à ce flux

 Les layouts : mettre en forme les messages

- Ces composants représentés par la classe org.apache.log4j.Layout permettent de définir le format du fichier de log. Un layout est associé à un appender lors de son instantiation.

Motif	Role
%C	le nom de la classe qui a émis le message
%d	le timestamp de l'émission du message
%m	le message
%n	un retour chariot
%L	Le numéro de ligne dans le code émettant le message : l'utilisation de ce motif est coûteuse en ressources
%l	Des informations sur l'origine du message dans le code source (package, classe, méthode)

■ Pour utiliser log4j, il y'a trois étapes à suivre :

1. Il faut ajouter le fichier log4j.jar dans le pom.xml de notre application
2. Il faut créer un fichier log4j.xml dans le src/main/resources qui contient la configuration de log4j pour l'application
3. Dans le code, il faut Obtenir une instance du logger relative à la classe

Séance 3 Junit :

Junit ça permet de **faire les tests**,

• Il existe différents niveaux de test :

– **Test d'intégration** : c'est le faite assembler plusieurs composants logiciel élémentaire

Ça permet de vérifier que ces composants élémentaires ça semblent parfaitement

– **Test de régression** : sont les tests exécutés sur un programme préalablement testé mais qui a subi une ou plusieurs modifications.

– **Test de charge (de capacité)** : s'agit d'un test au cours du quelle on va simuler un scénario bien précis

– **Test fonctionnel**

– **Test sécurité** : permet de découvrir la vulnérabilité du système,

– **Test unitaire :**

Un test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel. Il s'agit d'un code

Pour chaque classe (myclasse) , on a une classe de test (myclassTest) par convension

Il doit être indépendant, déterministe, isolé testé une fonction bien minimaliste, on teste un bout de code

- ❖ L'objectif de ces tests est de les développée et les automatisée

L'avantage :

- ❖ Garantie la non régression,
- ❖ détecter des bugs
- ❖ isoler les fonctions,
- ❖ voir l'avancement d'un projet (TDD)

Test-Driven development : le fait d'ajouter une fonctionnalité vide

Le fait d'écrire le test avant d'écrire la fonction

Au sein des tests, on utilise des assertions pour valider ou non un test

• JAVA → JUnit • SQL → SQLUnit • JS → JUnit PHP • → PHPUnit SimpleTest

Un test unitaire peut renvoyer 3 résultats :

- Success
- Error
- Failure

MOCK : on veut tester la création de la commande saisir correctement et on n'a pas encore développé la partie front, on simule le résultat qu'on va envoyer, on a besoin d'un retour d'un web service, on prépare les bouchons (les données mortes) et on peut tester

Séance 4 Injection de Dépendances :

On a 2 notions importantes :

- inversion de control IOC : est un design pattern, Ce principe-là n'est plus sous le contrôle de l'application, c'est le rôle du Framework spring
- Injection de dépendances ID : c'est la forme principale de l'IOC, c'est une méthode pour instancier des objets à travers `@Component` et créer les dépendances à travers `@Autowired` sans avoir de coder cela nous-même
 - ❖ Just on a ajouté les annotations, ce qui permet de réduire le code et sera plus lisible, et on gagne le temps de développement.

Bean : c'est une classe java

Bean spring : des classes avec des annotations

- **singleton** : Une seule instance du Bean créée pour chaque Conteneur IoC Spring, **par défaut**.
- **prototype** : Nouvelle instance créée à chaque appel du Bean
- **request** : (Contexte Web uniquement) Nouvelle instance créée pour chaque requête HTTP.
- **session** : une instance du bean par session HTTP.
- **global-session** : une instance du bean par session globale

Spring utilise les annotations pour indiquer quelle sont les beans qui travaillent avec lesquelles

Pour indiquer que c'est un bean spring on utilise **@component** : c'est une annotation générique

- Les beans de la couche **@Controller** (couche qui communique avec la partie front)
- Les beans de la couche Service : **@Service** (couche métier)
- Les beans de la couche DAO/Persistente : **@Repository** (données)
- ❖ `@Controller`, `@Service` et `@Repository` héritent de la classe `@Component`

Séance 5 : Spring boot :

Quelles sont les avantages de spring boot :

- ❖ On a un **serveur embarqué** dans le livrable, donc pas besoin de configurer un serveur externe
- ❖ Spring Boot facilite la gestion des configurations (**Auto configuration**) + configuration centralisé dans un seul fichier (application.properties)
- ❖ Spring Boot facilite la **gestion des dépendances** grâce au starter

Starters : permettre de télécharger un ensemble de dépendances

Mysql Driver

Starter-data-jpa : l'accès à la base de données

Starter-web : aidé à créer les services Rest ... exposer les web services

Spring boot devtools : grace à devtools, à chaque fois je fais des modifications et j'enregistre, automatiquement il va redémarrer le serveur

ERREUR : The Tomcat connector configured to listen on port 8080 failed to start. The port may already be in use or the connector may be misconfigure

- ❖ Il faut arrêter le Process Tomcat qui est en cours et relancer

RQ :



New Spring Starter Project

JSONException: A JSONObject text must begin with '{' at character 0



Service URL

No content available.

Si vous trouvez cette erreur quand tu créer un nouveau projet, donc il faut ajout « s » au http
@SpringBootApplication = @componentScan + @configuration +@enableAutoConfiguration

```
springBootProject/pom.xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ir
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
```

Une erreur dans la 1ère ligne du pom.xml

- ❖ Il faut ajouter : `<maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>`

Séance 6 : SPRING DATA JPA – PREMIÈRE ENTITÉ :

Persistence : sauvegarder les données dans le modèle relationnel (SGBDR)

Pour persister les données on a besoin : JDBC Ou bien Framework Data JPA :

✚ **JDBC** : interface qui va nous offrir les fonctionnalités de bases (create,update,delete) API pour les programmes utilisant JAVA, dans notre cas on ajoute MYSQL CONNECTER

- **Inconvénients** : – Pas de séparation entre le code technique et le code métier.
– la maintenance des requêtes à travers le JDBC est complexe

Donc pour éviter tous ces inconvénients : l'une des solutions, c'est l'utilisation de :

ORM (object relational Mapping) : pour éviter la complexité du code (Connexion + CRUD +Déconnexion), le mélangeant du code métier et les requêtes, l'idée de ORM est d'ajouter une couche qui va faciliter le travail Permet d'associer une ou plusieurs classes avec une table, et chaque attribut de la classe avec un champ de la table

EN JAVA : l'ORM c'est HIBERNATE

JPA (Java Persistence API) : est une interface de programmation Java permettant de normaliser l'utilisation et la communication avec la couche de données Est une spécification (normalisation et standardisation de la communication avec la DB).

Hibernate : est un produit (Implémentation de cette spécification).

Framework permet de gérer le mapping entre les objets de l'application et la base de données (plus simple)

Lorsqu'on travaille avec spring : il nous a fournis des outils pour améliorer la solution existante avec hibernate

SPRING DATA : permettre d'écrire plus simplement l'accès aux données, et ça va nous permettre comme hibernate d'accéder à la fonctionnalité du CRUD mais avec un code plus simple

Spring DATA : nous a offert 3 interfaces : **Repository , CrudRepository, PagingAndSortingRepository**

RQ : PagingAndSortingRepository = CrudRepository +autres fonctions (find avec sort) ...

On a aussi **JpaRepository** extends du PagingAndSortingRepository

SPRING DATA JPA : c'est un sous projet du Spring DATA

SÉRIALISATION / DÉSÉRIALISATION :

SÉRIALISATION : processus de conversion d'un objet en un flux d'octets pour le stocker ou l'envoyer.

ENTITÉ JPA :

@Table (name= "T_user") : facultatif, si on le met la table sera le même nom de l'entité

@Entity : Obligatoire, sur la classe : comme étant persistante, associée à une table dans la BD

@Id : La déclaration d'une clé primaire est obligatoire. Sur un attribut ou sur le getter

@GeneratedValue : Facultatif, sur l'attribut ou sur le getter annoté avec @Id. Définit la manière dont la base gère la génération de la clé primaire :

✚ @GeneratedValue (strategy = GenerationType.IDENTITY) : AUTO-INCREMENT (MYSQL)

✚ @GeneratedValue (strategy = GenerationType.Auto) : hibernate définit une séquence

✚ @GeneratedValue (strategy = GenerationType.Table) : hibernate_sequences

✚ @GeneratedValue (strategy = GenerationType.Sequence) : nous qui définir la sequence

@Column : est une annotation utile pour indiquer le nom de la colonne dans la table

- ✚ @Column(name=""EMPL_ID") : indique le nom de la colonne dans la table;
- ✚ @Column(length =) : indique la taille maximale de la valeur de la propriété;
- ✚ @Column(nullable= false) indique si la colonne accepte ou non des valeurs à NULL;
- ✚ @Column(unique=) : indique que la valeur de la colonne est unique

@Transient : Facultatif, sur un attribut

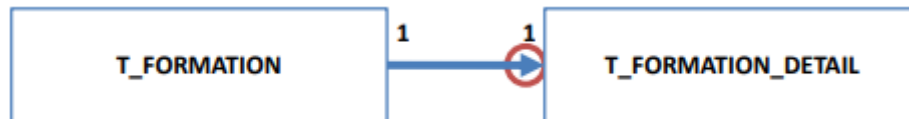
Indique que l'attribut ne sera pas mappé (et donc non persisté) dans la table

@Temporal : gère l'attribut en tant que date.

- ✚ @Temporal (TemporalType.DATE) : 30-09-19
- ✚ @Temporal (TemporalType.TIME) : 30-09-19 10:50:56.780000000 AM
- ✚ @Temporal (TemporalType.TIMESTAMP) : 1569840656 (nbre de secondes entre 01/01/1970 et la date voulue)

Séance 7 : SPRING DATA JPA – ASSOCIATIONS:

❖ One To One Unidirectionnelle :



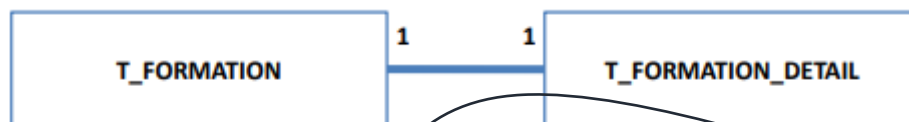
Dans la table formation :

@OneToOne

private FormationDetail formationDetail;

→ Ajoutant clé étranger dans formation

❖ One To One Bidirectionnelle



Dans la table formation :

@OneToOne

private FormationDetail formationDetail;

Dans la table formation_Detail :

@OneToOne(mappedBy="formationDetail")

private Formation formation;

RQ : Dans les relations BIDIRECTIONNELLE : il faut ajouter l'attribut "mappedBy" au niveau de fils, Qui permet de définir les deux bouts de l'association (Parent/child)

LA différence entre one to one unidirectionnel et one to one bidirectionnelle est de faire le mapping pour distinguer qui est le parent et qui est le child

→ La même que unidirectionnelle (Ajoutant clé étranger dans formation)

❖ One To Many Unidirectionnelle



Dans la table TP :

@OneToMany(cascade =CascadeType.ALL)

private Set <TpCorrection> TpCorrections;

→ Résultat : Table de jointure (id tp , id tpCorrection)

RQ : One To Many Bidirectionnelle = Many To One Bidirectionnelle :

❖ Many To One unidirectionnelle



Dans la table tpCorrection :

@ManyToOne(cascade = CascadeType.ALL)
TravauxPratiques travauxPratiques

❖ Many To One Bidirectionnelle



Dans la table tpCorrection :

@ManyToOne
TravauxPratiques travauxPratiques ;

Dans la table tp :

@OneToMany(cascade = CascadeType.ALL, mappedBy="travauxPratiques")
private Set TpCorrections;

RQ : L'attribut mappedBy est défini pour l'annotation @OneToMany, mais pas pour l'annotation @ManyToOne, car elle est toujours liée au «Child».

→ Donc on ne conclut que le mappedBy : toujours dans les relations bidirectionnel et dans la Table fils

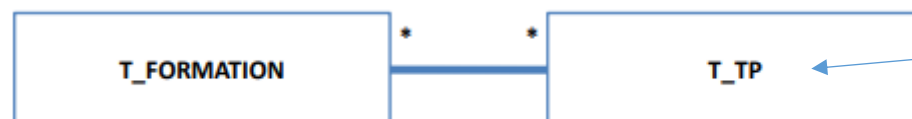
❖ Many To Many Unidirectionnelle



Dans la table formation :

@ManyToMany(cascade=CascadeType.ALL)
private Set formationTps;

❖ Many To Many Bidirectionnelle



fils

Dans la table formation :

@ManyToMany(cascade=CascadeType.ALL)
private Set formationTps

Dans la table tp :

@ManyToMany(mappedBy="formationTps", cascade = CascadeType.ALL)
private Set formations;

→ Toujours avec many To Many on ira une table de jointure

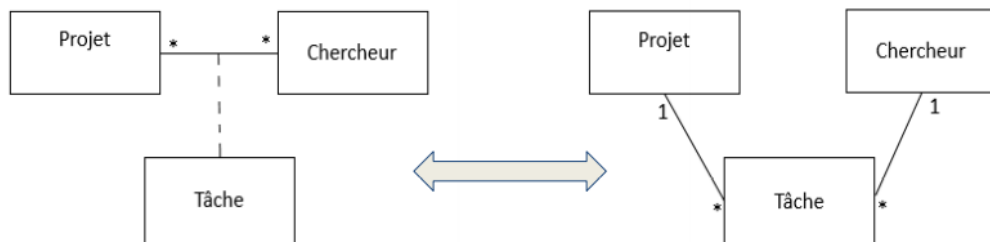
Les cas où on a besoin d'une table de jointure : 3

- One to Many Unidirectionnelle
- Many to Many Unidirectionnelle
- Many to Many Bidirectionnelle

Table porteuse de données :

Une association many to many est dite porteuse de données si la classe associative comporte des données autres que sa clé primaire

- Exemple : Une tâche est caractérisée par un projet, un chercheur et un nom.



Séance 8 : SPRING DATA JPA (CrudRepository) + JUnit:

Spring Data JPA est un sous projet du projet Spring Data

Avec spring DATA JPA , il suffit de définir une interface qui permet de manipuler une entité :

On a 3 interfaces : CrudRepository , JpaRepository , PagingAndSortingRepository

@Repository

public interface UserRepository **extends** JpaRepository<User, Long>

l'entité

Type de clé primaire

Donc avec cette ligne on accède aux méthodes du CRUD sans les écrire,

Mais si on veut faire des requêtes bien spécifiques, on a des Keyword : exemple

Keyword	Sample	Equivalent to
GreaterThan	findByAgeGreaterThan(int age);	Select u from User u where u.age > :age
LessThan	findByAgeLessThan(int age);	Select u from User u where u.age < :age
Between	findByAgeBetween(int from, int to);	Select u from User u where u.age between :from and :to
IsNull, NotNull	findByFirstnameNotNull();	Select u from User u where u.firstname is not null

On les ajoute au niveau de Repository

@Autowired

UserRepository userRepository;

→ On a injecté dans la couche service l'interface de la couche Repository, cela nous permet de profiter des méthodes qui existent dans l'UserRepository

Séance 8 : SPRING DATA JPA – JPQL

JPQL = Java Persistence Query Language

➔ Version orientée objet de SQL

Au lieu de manipuler les noms des tables on va manipuler les noms des entités JAVA

Et au lieu des nom des colonnes ➔ les attributs

@Modifying

- Ces méthodes permettent de récupérer les utilisateurs qui ont un role donné :

- **JPQL :**

```
@Query("SELECT u FROM User u WHERE u.role= :role")  
List<User> retrieveUsersByRole(@Param("role") Role role);
```

C'est équivalent à :

```
@Query("SELECT u FROM User u WHERE u.role= ?1")  
List<User> retrieveUsersByRole(Role role);
```

- **Native Query (SQL et non JPQL) :**

```
@Query(value = "SELECT * FROM T_USER u WHERE u.role= :role " , nativeQuery =  
true)
```

```
List<User> retrieveUsersByRole(@Param("role") Role role);
```

C'est équivalent à :

```
@Query(value = "SELECT * FROM T_USER u WHERE u.role= ?1 " , nativeQuery = true)  
List<User> retrieveUsersByRole(Role role);
```

RQ : pour l'insertion on ne peut pas utiliser **JPQL**

Séance 9 : SPRING MVC

DES REMARQUES :

- ✓ Pour DATE et LIST ➔ java.util
- ✓ Pour les ANNOTATIONS (@Entity , @Id , @Table ..) ➔ javax.persistence
- ✓ Si je mets pas @GeneratedValue.IDENTITY ➔ ça sera AUTO par défaut
- ✓ **MappedBy ➔ au niveau de fils = l'entité qui a le faible cardinalité**
- ✓ Repository ➔ La couche qui communique directement avec la base de données
- ✓ Pour consommer les fonctions des repository ➔ dans la couche service on injecte le Repository avec l'annotation @Autowired
- ✓ Différence entre LIST et OPTIONNEL ➔ si on n'a pas des résultats, LIST retourne nul, et OPTIONNEL retourne VIDE.
- ✓ 3 types de fonctions qu'on peut trouver dans la repository :
- ✓ --- 1ère méthode : on implémente l'interface JpaRepository/ CrudRepository
- ✓ --- 2ème méthode : on se base avec les keywords, on écrit des méthodes simples
- ✓ --- 3ème méthode : des requête bien spécifique avec JPQL @query (insert non)

La couche Controller : on va consommer les fonctionnalités qu'on a créées dans la couche service

DispatcherServlet joue le rôle du "Front Controller"

server.servlet.context-path=/SpringMVC

spring.mvc.servlet.path=/servlet

⇒ Grace à 2 lignes : on a construire le PATH

@ResponseBody // puisque la reponse en jSon

public User retrieveUser(@PathVariable("user-id") String userId)

⇒ @PathVariable : la fonction va chercher le path dont le path avec le paramètre user-id

Afficher → @GetMapping

Ajout → @PostMapping

Donc dans l'ajout, pour ne pas envoyer un path variable, on envoi tout un objet avec l'annotation **@RequestBody**

public User addUser(@RequestBody User u)

Suppression → @DeleteMapping

Modification → @PutMapping

RQ :

- Dans GET pour afficher un seul utilisateur, et dans Delete pour effacer un seul utilisateur → on utilise **@PathVariable**
- Dans POST pour ajouter un utilisateur en tant qu'un objet et dans @Put pour modifier un utilisateur en tant qu'un Objet → on utilise **@RequestBody**

Séance 10 : Introduction JSF Projet JSF & Spring

Vues : interfaces (client)

- ✚ L'architecture de notre projet tp-spring est : 3 tiers :
 - le client est Postman
 - la partie base de données : mySQL
 - l'intermédiaire : spring

Aujourd'hui on va travailler avec JSF non pas postman : toujours 3 tiers

- ✚ On a 2 types de Design pattern :
 - MVC1
 - MVC2

servlet avant contient : une méthode doPost , et une méthode doGet

MVC1 : chaque page web a un seul contrôleur

- ⇒ Donc il faut passer à une architecture plus développée → **MVC2** : jsf
- ⇒ L'objectif de JSF : est de gérer plusieurs contrôleurs à la fois

On ouvre Project Explorer : car dans package explorer je n'ai pas le webApp

Le Project Explorer plus claire en affichage

Avec JSF on utilise @Controller

- On a pas une technologie de vie propre à spring , par contre il peut être compatible avec d'autre Framework WEB , donc pour cela dans notre projet in a jouté nous-même la structure (webapp ...) et on va configurer le projet pour qu'il se porte JSF , et aussi au niveau des dépendance , on les ajoute nous même
- Une requête JPQL ne porte pas "VALUE " et "NativeQuery = true "
- **@RestController** : pour la consommer avec postman ou les exposer.
- Ici avec JSF communication dans le même projet → **@Controller**
- **@ELBeanName** : faire la correspondance entre le contrôleur et page JSF (liaison)

Séance 11: JSF + Spring (CrudRepository)

Séance 12 : JSF (Filter Converter Validator) + Spring

✚ Sécuriser une méthode

⇒ Problème de sécurité, n'importe qui peut accéder au CRUD sans s'authentifier. Comment sécuriser toutes nos méthodes (CRUD) :

- 1- On ajoute chaîne de caractère "String navigateTo = "null";" et la condition

```
public String addEmployee() {  
    String navigateTo = "null";  
    if (authenticatedUser==null || !loggedIn) return "/login.xhtml";  
    employeeService.addOrUpdateEmployee(new Employee(nom, prenom, email,  
        password, actif, role));  
    return navigateTo;  
}
```

Mais **ATTENTION** : ce n'est pas suffisant, la solution la plus pratique, il faut sécuriser :

✚ – Sécuriser une page (FILTER)

- 1- Créer le package tn.esprit.spring.config : configurer l'application
- 2- Créer la classe LoginFilter qui implémente javax.servlet.Filter

```
package tn.esprit.spring.config;  
  
import javax.servlet.Filter;  
...  
  
public class LoginFilter implements Filter {  
  
    @Override  
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,  
        FilterChain filterChain) throws IOException, ServletException {  
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;  
        HttpServletResponse httpServletResponse = (HttpServletResponse) servletResponse;  
        ControllerEmployeeImpl employeeController = (ControllerEmployeeImpl)  
            httpServletRequest.getSession().getAttribute("employeeController");  
        if (employeeController!=null && employeeController.getAuthenticatedUser() != null &&  
            employeeController.getLoggedIn()) { filterChain.doFilter(servletRequest, servletResponse);}  
        else {httpServletResponse.sendRedirect(httpServletRequest.getContextPath() +  
            "/Login.jsf");}  
    }  
}
```

Instance de EmployeeImpl qu'on a créer et récupérer l'attribut

Si l'objet n'est pas null et bien authentifier donc l'orchestrateur permettre de passer d'une page à une autre

Sinon il va de rediriger de nouveau vers le login ;):p

- 3- Indiquer à Spring qu'il doit charger ce bean au démarrage (le bean de configuration qu'on a déjà créer) : au niveau de notre classe main :

```
@Bean  
public FilterRegistrationBean loginFilter() {  
    FilterRegistrationBean registration = new FilterRegistrationBean();  
    registration.setFilter(new LoginFilter());  
    registration.addUrlPatterns("/pages/*");  
    return registration;  
}
```

Une fois qu'on a sécurisé les méthodes, les pages, la dernière étape qui nous reste c'est la faire de **valider le formulaire** par les **CONVERTER** :

On a deux types de **CONVERTER** :

✚ **Implicit** : il peut convertir tout seul implicitement (coté développement j'ai rien à faire)

• **Implicit converter** :

- FloatConverter - For Conversion between String and java.lang.Float
- BooleanConverter - For Conversion between String and java.lang.Boolean
- IntegerConverter - For Conversion between String and java.lang.Integer

- Dans la JSF :

```
<h:inputText value="#{employeeController.salaire}"></h:inputText></td>
```

- Dans le Contrôleur :

```
private float salaire;
```

Mais d'autre exemple : comme le **date**, il faut explicitement indiquer comment il va faire la convention

✚ **Explicit** : avec des validator

○ 1ère exemple

```
<h:inputText id="date" required="true" requiredMessage="Date Début  
Obligatoire"  
value="#{employeeController.dateDebut}"  
<f:convertDateTime pattern="dd-MM-yyyy" />  
</h:inputText>
```

cad ne le valider
sauf qu'il est sous
cette format

○ 2ème exemple

```
<h:message for="validateSalaire" style="color:green" />  
<h:inputText id="validateSalaire"  
value="#{contratBean.salaire}"  
required="true"  
requiredMessage="Le contrat doit spécifier le salaire"  
validatorMessage="Le salaire doit être compris entre 350 et 3500">  
<f:validateDoubleRange minimum="350" maximum="3500" for="validateSalaire"/>  
</h:inputText>
```

Je veux un salaire est obligatoire,

○ 3ème exemple

```
<h:message for="validationEmail" style="color:green" />  
<h:inputText id="validationEmail"  
value="#{employeeBean.email}"  
required="true"  
requiredMessage="Le champs email est obligatoire">  
<f:validateRegex pattern=".+@.+.+.+" />  
</h:inputText>
```

Le message d'erreur généré par l'inputText va être affiché
dans le <h:message/>.

pour le mail est obligatoire

pour les pattern

Séance 13 : SPRING – AOP

- AOP : Aspect Oriented Programming, ou Programmation Orientée Aspect

⇒ FRAMEWORK ASPECT.js

Mais nous avec spring intègre l'ASPECT.j avec spring AOP

- Permet de rajouter des comportements à des classes ou des méthodes existantes

- Ajouter des traces (logs)
- Ajouter la gestion des transactions
- Ajouter la gestion de la sécurité
- Ajouter du monitoring

- AOP peut être une solution à ces problèmes **Cross Cutting Concerns:**

– **Tangling** : Mélange du code métier avec du code technique

– **Scattering** : Duplication d'un bout de code dans plusieurs endroits

- Pour corriger les problèmes de **Cross Cutting Concerns**

- **Tangling** → **SoC** : Séparer le code technique et le code métier
- **Scattering** → **DRY** : Eviter la redondance au niveau du code (duplication)

- Avec nos starter JPA et WEB on a installé déjà l'AOP

- Les AVANTAGES AOP :

- Facilité de maintenance
- Permet une meilleure modularité et la réutilisation

- Les INCONVINEANTS AOP :

- Nécessite un temps de prise en main.
- La lecture du code contenant les traitements ne permet pas de connaître les aspects qui seront exécutés

- L'AOP peut être utilisée :

- DIRECTEMENT : Avec les ASPECTS @After , @Befor ..
- INDIRECTEMENT : avec @Configuration, @Transactionnel

@Transactionnel : ou bien tu fais tout le traitement ou bien tu fais rien (tout ou rien)



L'AOP utilise le Design Pattern **Proxy**.

- Implémentation d'AOP :

- Pointcut : Une expression, qui permet de sélectionner plusieurs Join points.
- Join point : Toutes les méthodes de services (elle se trouve dans le service)
- Advice : Le code que l'on veut rajouter (ce qu'il existe dans la méthode)
- Aspect : composée d'un ou de plusieurs Pointcut et Advice. La classe est annotée @Aspect.
- Weaving : le faite d'insérer des aspects

```

@Component
@Aspect
public class LoggingAspect {           Aspect (toute la classe)

private static final Logger logger = LogManager.getLogger(LoggingAspect.class);

@Before("execution(* tn.esprit.esponline.service.*(..))") Pointcut
public void logMethodEntry(JoinPoint joinPoint) {
    String name = joinPoint.getSignature().getName();
    logger.info("In method " + name + " : ");
}

```

Advice

Type
d'ADVICE

- @Before
- @After : s'exécute après dans tous les cas
- @AfterReturning : il va afficher si la méthode s'exécute correctement
- @AfterThrowing : s'exécute après une exception
- @Around : s'exécute autour du join point

■ Pour intégrer l'AOP dans mon projet :

1- Au niveau de la classe main

```

@SpringBootApplication
@EnableAspectJAutoProxy           Activation du Spring AOP
public class SpringBootDataJpaMvcJspApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootDataJpaMvcJspApplication.class, args);
    }
}

```

RQ : la classe main est connu avec @SpringBootApplication

2- Séparer le code métier et le code technique, Exemple je veux ajouter un logging dans toutes les classes de service, le temps de calcul.

→ Donc on ajoute un package config ou je vais faire la configuration de l'AOP et je traite notre classe : Exemple :

```

package tn.esprit.esponline.config;
@Component           il va le charger dans IOC Container
@Aspect             indique que cette classe est un ASPECT, ell va ajouter la programmation orientée ASPECT
public class LoggingAspect {
    private static final Logger l = LogManager.getLogger(LoggingAspect.class);
    @Before("execution(* tn.esprit.esponline.service.UserServiceImpl.*(..))")
    public void logMethodEntry(JoinPoint joinPoint) {
        String name = joinPoint.getSignature().getName();
        logger.info("In method " + name + " : ");
    }
    @After(".....")
    public void logMethodExit...
}

```


ADVICE :

- `"execution(Modifiers-pattern? Ret-type-pattern Declaring-type-pattern?Name-pattern(param-pattern) Throws-pattern?)"`
- "?" veut dire optionnel
- **Modifiers-pattern?** : public, private ...
- **Ret-type-pattern** : le type de retour.
- **Declaring-type-pattern?** : nom de la classe y compris le package.
- **Name-pattern** : nom de la methode.
- **Throws-pattern?** : l'exception.
- ".." veut dire, 0 ou plusieurs paramètres

Exercice Advise

Expliquer les PointCut suivants :

- `@Before("execution(* tn.esprit.esponline.service.*.*(..))")`

⇒ Tous les méthodes qui existe sous le package service

- `@Before("execution(public *.*(..))")`

⇒ Tous les méthodes public n'importe qu'elle package (mais c'est dangereux cette méthode, elle va bloquer l'application)

- `@Before("execution(* set*(..))")`

⇒ N'importe quelle méthode qui commence par **Set**

- `@Before("execution(* tn.esprit.esponline..*.*(..))")`

⇒ N'importe quelle package