**Application générale sur spring :**

**Les annotation Jpa :** ces annotations sont appliquées sur des classes du package model, pour que spring puisse après créer des tables qui leurs correspondant au niveau de base de donnée**.**

**@Entity:** Utiliser sur une classe pour dire á Jpa de crée une table qui lui correspond dans la base de données.

**@Table(name = ‘’tableName’’)** utiliser sur une classe pour spécifier le nom de la table dans la base de données .

**@DateTimeFormat(pattern =** "yyyy-MM-dd"**)** pour spécifier le format de la date.

**@Temporal(TemporatType.DATE)** utiliser sur une propriété de type date pour se contenter d’afficher que la partie jour-mois-année.

**@NotNull** utiliser sur une propreté de la classe pour quelle n’accepte pas une valeur null.

**@Id** **@GeneratedValue(strategy = GenerationType.AUTO)** utiliser sur une propriété pour dire qu’il s’agit de l’identifiant de la table et qu’il est auto incrémenter.

Relation ManyToMany :

Utiliser lorsqu’on a 2 listes de 2 classes différente qui sont en relation entre eux, dans la base de données cette relation sera traduit par une table d’association dont la clé primaire est la concaténation des identifiants de 2 classes :

Exemple : un employé appartient á un groupe ou plusieurs, et un groupe contient 0 ou plusieurs employées, ceci est traduit par une liste des employés et une liste des groupes respectivement dans la classe groupe et la classe employé.

@ManyToMany @JoinTable(

name = "EMP\_GRP",

joinColumns = @JoinColumn(name = "CODE\_EMP"),

inverseJoinColumns = @JoinColumn(name = "CODE\_GRP")

)

**private** Collection<Groupe> groupes;

EMP\_GRP sera le nom de la table d’association, et que joinColumns et inverseJoinColumns pour renommer la clé primaire de la table d’association qui est la concaténation d’identifiants des 2 classes.

@ManyToMany(mappedBy = "groupes") Collection<Employe> employés 🡪 dans l’autre propriété il suffit de dire que la mapping a été faite par la propriété de l’autre classe.

Relation OneToMany et ManyToOne :

Utiliser lorsqu’on a une liste d’une classe qui est en relation avec une propriété simple d’une autre classe, la liste sera annoter par @OneToMany, et la propriété sera annoter par @ManyToOne, cela est traduit par une clé étrangère dans la table qui contient la propriété simple :

Exemple : un client peut avoir 0 ou plusieurs comptes ce qui est traduit par une liste des comptes, et un compte appartient á un ,et un seul client ce qui est traduit par une propriété compte:

@ManyToOne @JoinColumn(name = "CODE\_CLIENT") **private** Client client; 🡪 utiliser coté classe compte, @JoinColumn pour ajouter dans la table compte la colonne CODE\_CLIENT qui fait référence á l’identifiant de la table client.

@OneToMany(mappedBy = "client") **private** Collection<Compte> comptes; 🡪 coté de la classe client, ici "client" pour dire que le mapping est faite par la propriété appelée client de la classe compte.

Mapper l’héritage dans la base de donnée :

La classe mère doit avoir l’annotation @Inheritance ou on spécifier la stratégie de l’implémentation d’héritage coté base de données qui est ici l’utilisation d’une seule table qui va contenir toutes les classes en utilisant un champ indiquant le type de la classe.

**@Inheritance(strategy = InheritanceType.*SINGLE\_TABLE*)** utiliser sur la classe mère pour indiquer l’héritage, la stratégie sera de regrouper toutes les classes (classes mère et classes filles) dans une seul table.

**@DiscriminatorColumn(name = "type\_compte",discriminatorType = DiscriminatorType.*STRING*,length = 2)** utiliser également sur la classe mère ,indiquant le nom et le type de la colonne qui va mapper l’héritage.

**@DiscriminatorValue("CC")** utiliser sur les classes filles, pour dire qu’un enregistrement dans la colonne qui mappe l’héritage qui a la valeur **("CC")** correspond á cette classe fille.

Déclaration d’une contrainte unique sur multiple propriétés :

**@Table( name="EMPLOYEE", uniqueConstraints= @UniqueConstraint(columnNames={"EMP\_ID", "EMP\_NAME"}) )** ici , on dit que la concaténation de emp\_id et emp\_name est une clé unique.

Annotation de validation des champs :

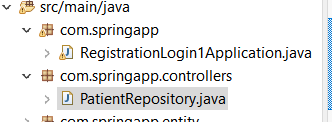
**@Size(min = 4, max = 32)** test si un champ de type String et de taille minimum sup á 4, et ne dépasse pas 32.

**@Email(message="Please provide a valid email address")** pour exiger d’entre un email valide.

Pour afficher le message d’erreur qui correspond á les annotations de validation sous thymealef en utilise :

**<span th:errors=*"${patient.id}"*></span> 🡪 id entrée est null;**

**Spring data: utiliser pour effectuer les différentes opérations CRUD sur une table qui est déjà annoter par @Entity, pour y faire il faut crée une interface annoter par @Repository qui extend l’interface générique JpaRepository en lui passant 2 paramètres le nom de la classe, et le type de l’identifiant de la classe.**



Dans la couche metier (classe metier) en instance l’objet de l’interface repository á l 'aide de l’annotation @Autowired, ce qui permet d’appeler des méthodes prédéfinit par spring data :

objetRepository.**save(Objet objet)** pour enregistrer ou bien mettre á jour l’objet dans la base de donnée.

objetRepository.**findAll()** retourne un tableau List<Objet> qui contient tous les enregistrements de la table.

objetRepository.**findById(Id id)** retourne un objet optional, qui va contient soit l’objet soit null.

objetRepository.**findByAttribute()** chercher par une propriété spécifique de l’objet, retourne une liste d’objets.

On peut aussi créer notre propres méthodes de l’accès á la base de données, cela est fait par l’annotation @Query qui nous donne la possibilité d’écrire une requête utilisant sql, ou bien en utilisant hql :

Les requêtes sql exigent l’option nativeQuery = true ;

@Query(value = "Select \* from patient where name like ?1 or last\_name like ?1",nativeQuery = **true**)

List<Patient> findByNameOrByLastNameLike(String keyword);

* ?1 correspond au 1ere paramètres de la méthode, ?2 le cas où la méthode contient un 2eme paramètres.

**Les requêtes hql utilise les classes et les relations des classes, plutôt que les tables et les relations des tables :**

@Query("Select o From Operation o where o.compte.codeCompte=:x order by o.dateOperation desc")

List<Operation> listOperation(@Param("x")String codeCompte) ;

:x spécifier l’emplacement ou sera injecter le paramètres de la méthode, á condition que ce paramètres soit annoter par @Param(‘x’)🡪 cette requête équivalente en sql á select \* from operation where code\_compte = ? order by date\_operation des ;

@Query("SELECT e FROM Employe e inner join e.groupes groupe WHERE groupe.codeGroupe=:x") 🡪 cette requête exprime une jointure interne, permet ici de chercher les employés qui appartient á un groupe x, la recherche se fait par trouvé le groupe x dans la liste des groupe pour chaque employé, la classe employé contient attribut groupe d’où inner join e.groupes dans laquelle on chercher sur le groupe x ;

On a utilisé cette requête pour sélectionner que l’id et docNam de l’objet Doc dans une classe qui contient que ces 2 propriétés, á raison que la sélection de l’objet Doc entière cause des problèmes de performance.

@Query("SELECT new net.javaguides.model.FileInfo(d.id , d.docName) FROM Doc d where d.dossierMedical.id =:x")

List<FileInfo> findAllDocsByCodeDossier(@Param("x")Long codeDossier);

**Spring MVC(WEB) :**

D'abord en crée un package des contrôleurs dans lequel on va y avoir tous les contrôleurs de l’application, un contrôleur est une classe annotée par @Controller.

Pour passer des éléments de la méthode vers la vue qui est une page html, on utilise l’objet Model **model.addAttribute(‘’nomVar’’,var).**

**Thymeleaf :** Pour utiliser thymleaf dans les pages html, il faut l’inclure avec le lient ci-dessous :

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

Quelque balise Thymeleaf pour traiter les données passer dans le model :

**<td th:text=*"${p.name}"* ></td>** th:text sert á afficher la valeur name de l’objet p á l’emplacement indiqué.

**<tr th:each=*"p:${patients}"* >** traitement </tr> utiliser comme une boucle qui à chaque fois met un objet de la liste patients dans la variable p jusqu’à parcourir toute la liste patients.

**<div th:if=*"${param.success}"*> contenu de div </div> 🡪 th :if = ‘{condition}’** si la conditions est vrais, alors le contenu de la balise div sera afficher, param.success vrais si l’URL contient un paramètres appeler success.

**<a th:href=*"@{patients(page=${status.index})}”* >**🡺 pour dire consulter la méthode annoter par /patient en lui passant le paramètres page qui a comme valeur ${status.index} 🡪 /patients?page=valeurX.

**<form th:action=*"@{regestration}"* method = *"post"* th:object=*"${user}"*> 🡪** th:action = @{registration} permet de dire que après la validation du formulaire va consulter la méthode du contrôleur regestration, concernant th :object = ‘{user}’ il s’agit de l’objet passer dans le model pour être remplie á l’aide du formulaire.

**<input type = *"text"* id = *"fistName"* class = *"form-control"* th:field = *"\*{firstName}"* >** 🡪 th:field = “\*{firstName}” pour dire que la valeur saisir dans cette balise input va être stocker dans l’attribut firstName de l’objet user qui est passer par th :object au niveau de la balise <form>.

**<th:text=*"${#httpServletRequest.getParameter('error')****}****"* >** permettre de récupérer la valeur d’un paramètre depuis la requête http .

Créé une page template :

Il s’agit d’une page html qui contient le contenu statique qui ne change pas au niveau de toutes les pages html de l’application, par exemple elle doit contient l’header le footer la barre de navigation etc….

Pour donc qu’une page soit une template dans thymleaf la balise html doit contenir 2 attributs xmlns :th et xmlns : layout

**<html xmlns:th= *"http://thymleaf.org"* xmlns:layout=*"http://www.ultraq.net.nz/thymeleaf/layout"*>**

**<head>**

**<meta charset=*"UTF-8"*>**

**<title>template</title>**

**<link rel = *"stylesheet"* href=*"/webjars/bootstrap/4.6.0/css/bootstrap.min.css"*/>**

**<script src=*"/webjars/jquery/3.4.1/jquery.min.js"*></script>**

**<script src=*"/webjars/bootstrap/4.6.0/js/bootstrap.min.js"*></script>**

**</head**>

Pour inclure cette layout dans les autres pages, il faut que pour chaque page on ajoute les 3 attributs ci-dessous á la balise html de la page :

**<html xmlns:th= *"http://thymleaf.org"* xmlns:layout=*"http://www.ultraq.net.nz/thymeleaf/layout"* layout:decorate=*"template1"*>** 🡪 layout:decorate permet de spécifier la page template.

La page layout doit contenir la balise <section layout :fragment= ‘’pageContent’’></section> dont le contenu sera vide, mais qui indique que cette partie du page présente la partie dynamique qui va être utiliser pour contenir le corps des autre pages html, et pour y faire on doit ajouter <div layout :fragment= ‘’pageContent>Contenu de la page</div> dans le body de la page qui souhaite utiliser la template.

**Lombok annotations :**

@Data pour générer les getters and setter

@NoArgsConstructor générer le constructeur sans paramètres

@AllArgsConstructor générer le constructeur avec tous les paramètres

@ToString pour redéfinir la méthode toString de la classe object

**Couche service :**

Dans la couche metier, Les classes d’implémentations doivent avoir ces 2 annotations @Service, @Transacional

Pour lever une exception de la couche metier dans java on utilise **throw** **new** RuntimeException("Opération impossible");

**Spring Security :**

La classe qui va gérer la sécurité de l’application doit être annoter par **@Configuration et @EnableWebSecurity** et d’étendre la classe **WebSecurityConfigurerAdapter** qui a 2 méthodes dont on doit redéfinir :

**http.authorizeRequests().anyRequest().authenticated();** pour dire a spring sécurité que toutes les requêtes nécessite l’authentification.

**http.formLogin().loginPage("/authentificationPage")** pour exiger l’utilisation de la page personnalisée d’authentification .

**http.formLogin()** pour utiliser la page d’authentification fournit par springSecurity.

**auth.inMemoryAuthentication().withUser("medcin1").password("{noop}1234").roles("medcin");** pour crée une authentification simple et statique dans la mémoire, {noop} pour dire de ne pas utiliser l’encoder

http.authorizeRequests().antMatchers("/addPatient/\*\*","/deletePatient/\*\*","/updatePatient/\*\*").hasRole("MEDCIN"); pour dire que ces pages nécessite le rôle médecin.

**auth.inMemoryAuthentication().withUser("medcin1").password(passwordEncoder().encode("1234")).roles("medcin");**

Au moment d’insertion de mot de passe il faut injecter l’objet retourner par notre méthode

**http.exceptionHandling().accessDeniedPage("/notAuthorized");** pour rediriger les accès non autorisés vers une page des erreurs.

Pour crypter le mot de passe on utilise une méthode qui est annoter par @Bean (pour qu’on puisse l’injecter dans toute l’application à l’aide de @Autowired) :

**@Bean**

**public PasswordEncoder passwordEncoder() {**

**return new BCryptPasswordEncoder();**

**}**

Pour gérer la contextualisation il faut ajouter la dépendance Thymeleaf extra spring security 5 et d’inclure l’entête suivante dans la page template de Thymeleaf :

**xmlns:sec=**[***http://www.thymeleaf.org/thymeleaf-extras-springsecurity5***](http://www.thymeleaf.org/thymeleaf-extras-springsecurity5)

**<a class=*"dropdown-item"* th:href=*"@{logout}"*>Login</a>** pour se deconnecter.

**sec:authorize = *"isAuthenticated*()"** utiliser sur une balise pour afficher son contenu, á condition que la session est établie. (!isAuthentidacted() pour dire l’inverse isAnonymous).

**<span sec:authentication= *"name"*></span>**permet d’afficher le nom de l’utilisateur courant.

**<div sec:authorize=*"hasRole('ROLE\_ADMIN') OR hasRole('ROLE\_MEDCIN')"* >** le contenu de div sera afficher á condition que l’utilisateur authentifié soit un admin ou bien un médecin.

@Configuration

@EnableWebSecurity

**public** **class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter{

@Autowired

**private** UserService userService;

@Bean

**public** BCryptPasswordEncoder passwordEncoder() {

**return** **new** BCryptPasswordEncoder();

}

@Bean

**public** DaoAuthenticationProvider authenticationProvider() {

DaoAuthenticationProvider auth = **new** DaoAuthenticationProvider();

auth.setUserDetailsService(userService);

auth.setPasswordEncoder(passwordEncoder());

**return** auth;

}

@Override

**protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception{

auth.authenticationProvider(authenticationProvider());

}

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.authorizeRequests().antMatchers("/registration\*\*","/js/\*\*","/css/\*\*","/img/\*\*").permitAll()

.anyRequest().authenticated().and().formLogin().loginPage("/login").permitAll().and().logout()

.invalidateHttpSession(**true**).clearAuthentication(**true**).logoutRequestMatcher(**new** AntPathRequestMatcher("/logout"))

.logoutSuccessUrl("/login?logout").permitAll();

}

}

**Pour faire adapter Spring Security d’avoir les résultats depuis la base de données, il faut utiliser UserDetailService :**

**public** **interface** UserService **extends** UserDetailsService{

User save(UserRegistrationDto registrationDto);

}

@Override

**public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {

User user = userRepository.findByEmail(username);

**if**(user == **null**) {

**throw** **new** UsernameNotFoundException("Invalid username or password");

}

**return** **new** org.springframework.security.core.userdetails.User(user.getEmail(),user.getPassword(), mapRolesToAutorities(user.getRoles()));

}

**private** Collection<? **extends** GrantedAuthority> mapRolesToAutorities(Collection<Role> roles){

**return** roles.stream().map(role -> **new** SimpleGrantedAuthority(role.getName())).collect(Collectors.*toList*());

}