Utilisation de Lombok et Spring Core

# Démonstration 5 du module 1

|  |
| --- |
| Les objectifs de cette démonstration :   * Rappel sur la création de bean avec Spring Core * Injection par constructeur avec Lombok |

Déroulement

# Contexte

* Continuer l’application précédente
* Notre application va se compléter au fil des démonstrations, en représentant une partie du cœur de métier de l’ENI Ecole.
* Dans cette itération, il faut créer :
  + Une classe bouchon pour la couche d’accès aux données : EmployeRepository
    - Elle va simuler une base de données avec une liste statique d’employés
    - Les méthodes du design pattern CRUD (Create Read Update Delete)
    - Et des find (findAll et findByImmatriculation)
  + Et une classe métier qui permet de les gérer :
    - Ajoutant un employé
      * En validant les données en amont
    - Retournant tous les employés
* La configuration du projet actuel correspond à cette itération

# Mise en place de Spring Core

## Dans la Couche DAL, déclarer un bean de Spring

* Création d’une interface EmployeDAO pour un couplage faible
  + Design pattern DAO = Data Access Object
  + Elle suit le design pattern CRUD = Create Read Update Delete
  + Elle ajoute 2 méthodes de recherche findByImmatriculation et findAll

**package** fr.eni.demo.dal;

**import** java.util.List;

**import** fr.eni.demo.bo.Employe;

**public** **interface** EmployeDAO {

**void** create(Employe employe);

Employe read(Integer id);

Employe findByImmatriculation(String immatriculation);

List<Employe> findAll();

**void** update(Employe employe);

**void** delete(Employe employe);

}

* Création de son implémentation
  + C’est une classe bouchon pour permettre de simuler la couche d’accès aux données des employés
    - Un attribut List<Employe> permet de gérer le stockage des informations
  + EmployeDAOImpl est annotée @Component pour être un bean de Spring

**package** fr.eni.demo.dal;

**import** java.util.\*;

**import** org.springframework.stereotype.Component;

**import** fr.eni.demo.bo.Employe;

@Component

**public** **class** EmployeDAOImpl **implements** EmployeRepository {

**private** List<Employe> employes = **new** ArrayList<>();

@Override

**public** **void** create(Employe employe) {

employes.add(employe);

}

@Override

**public** Employe read(Integer id) {

**return** employes

.stream()

.filter(item -> item.getId() == id)

.findAny()

.orElse(**null**);

}

@Override

**public** Employe findByImmatriculation(String immatriculation) {

**return** employes

.stream()

.filter(item -> item.getImmatriculation() == immatriculation)

.findAny()

.orElse(**null**);

}

@Override

**public** List<Employe> findAll() {

**return** employes;

}

@Override

**public** **void** update(Employe employe) {

Employe emp = read(employe.getId());

**if** (emp != **null**) {

emp.setEmail(employe.getEmail());

emp.setPrenom(employe.getPrenom());

}

}

//Il est important que les méthodes Equals et HashCode soient redéfinies pour cela

@Override

**public** **void** delete(Employe employe) {

employes.remove(employe);

}

}

## Dans la Couche BLL, déclarer un bean de Spring

* Création de l’interface EmployeService pour qu’il y ait un couplage faible :

**package** fr.eni.demo.bll;

**import** java.util.List;

**import** fr.eni.demo.bo.Employe;

**public** **interface** EmployeService {

**void** ajouter(Employe employe);

List<Employe> chargerTousEmployes();

}

* Création d’une implémentation EmployeServiceImpl
  + Utilisation de l’annotation @Service pour déclarer un composant métier
  + Utilisation de l’annotation de Lombok pour gérer la génération du constructeur avec le paramètre EmployeRepository
  + Cette classe fait un minimum de validation avant de créer un nouvel employé
    - Les chaines de caractères ne sont pas nulles ou vides
    - L’immatriculation est unique

**package** fr.eni.demo.bll;

**import** java.util.List;

**import** org.springframework.stereotype.Service;

**import** fr.eni.demo.bo.Employe;

**import** fr.eni.demo.dal.EmployeRepository;

**import** lombok.AllArgsConstructor;

//Permet de faire injecter la couche DAL associée

@AllArgsConstructor

@Service

**public** **class** EmployeServiceImpl **implements** EmployeService {

**private** EmployeRepository employeRepository;

@Override

**public** **void** ajouter(Employe employe) {

// Validation des données de l'employé avant sauvegarde

**if** (employe == **null**) {

**throw** **new** RuntimeException("L'employé n'est pas renseigné");

}

validerImmatriculation(employe);

validerChaineNonNulle(employe.getNom(), "Vous devez renseigner le nom");

validerChaineNonNulle(employe.getPrenom(), "Vous devez renseigner le prénom");

validerChaineNonNulle(employe.getEmail(), "Vous devez renseigner un email");

employeRepository.create(employe);

}

@Override

**public** List<Employe> chargerTousEmployes() {

**return** employeRepository.findAll();

}

**private** **void** validerChaineNonNulle(String chaine, String msgErreur) {

**if** (chaine == **null** || chaine.isBlank())

**throw** **new** RuntimeException(msgErreur);

}

**private** **void** validerImmatriculation(Employe employe) {

// Valider que l'immatriculation n'est pas nule ou vide

validerChaineNonNulle(employe.getImmatriculation(), "L'immatriculation n'a pas été renseignée");

// Immatriculation doit être unique

Employe employeDB = employeRepository.findByImmatriculation(employe.getImmatriculation());

**if** (employeDB != **null**) {

**throw** **new** RuntimeException("L'immatriculation doit être unique");

}

}

}

# Création de tests unitaires

* Pour valider les comportements de la couche métier
  + Création d’une classe de tests : TestEmpoyeService
  + Utilisation de Mockito pour gérer le bouchon de la couche DAL

**package** fr.eni.demo.bll;

**import** **static** org.assertj.core.api.Assertions.*assertThat*;

**import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;

**import** **static** org.junit.jupiter.api.Assertions.*assertNotNull*;

**import** **static** org.junit.jupiter.api.Assertions.*assertThrows*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.\*;

**import** org.junit.jupiter.api.\*;

**import** org.mockito.\*;

**import** fr.eni.demo.bo.Employe;

**import** fr.eni.demo.dal.EmployeDAO;

**class** TestEmployeService {

**private** EmployeService employeService;

@Mock// Injection d'un Mock du EmployeDAO

**private** EmployeDAO employeRepository;

@BeforeEach

**void** init() {

MockitoAnnotations.*openMocks*(**this**);

employeService = **new** EmployeServiceImpl(employeRepository);

}

@Test

**void** test01\_ajouter\_tousParametresValides() {

**int** id = 1;

Employe employe = Employe

.*builder*()

.id(id)

.nom("BAILLE")

.prenom("Anne-Lise")

.email("abaille@campus-eni.fr")

.immatriculation("ENI\_Ecole\_012892")

.build();

//Définir le comportement du Repository mocké

*when*(employeRepository.read(id)).thenReturn(employe);

//Comportemnet à valider

employeService.ajouter(employe);

// Vérification de l'ajout dans la liste des employés

Employe employeDB = employeRepository.read(id);

*assertNotNull*(employeDB);

*assertThat*(employe.getImmatriculation()).isEqualTo(employeDB.getImmatriculation());

*assertThat*(employe.getEmail()).isEqualTo(employeDB.getEmail());

*assertThat*(employe.getNom()).isEqualTo(employeDB.getNom());

*assertThat*(employe.getPrenom()).isEqualTo(employeDB.getPrenom());

*assertThat*(employe.getNumDom()).isEqualTo(employeDB.getNumDom());

*assertThat*(employe.getNumPortable()).isEqualTo(employeDB.getNumPortable());

}

// Employe null

@Test

**void** test\_ajouter\_employe\_null() {

*assertThrows*(RuntimeException.**class**, () -> employeService.ajouter(**null**));

}

// Nom nul - la validation métier l'interdit - nom est obligatoire

@Test

**void** test\_ajouter\_sansNom() {

Employe employe = Employe

.*builder*()

.id(2)

.prenom("Stephane")

.email("sgobin@campus-eni.fr")

.immatriculation("ENI\_Ecole\_012111")

.build();

*assertThrows*(RuntimeException.**class**, () -> employeService.ajouter(employe));

}

// Immatriculation vide ou nulle - la validation métier l'interdit

@Test

**void** test\_ajouter\_immatriculationVide() {

Employe employe = Employe

.*builder*()

.id(2)

.nom("GOBIN")

.prenom("Stephane")

.email("sgobin@campus-eni.fr")

.immatriculation("")

.build();

*assertThrows*(RuntimeException.**class**, () -> employeService.ajouter(employe));

}

// Immatriculation Unique - la validation métier l'interdit

@Test

**void** test\_ajouter\_immatriculationUnique() {

Employe employe1 = Employe

.*builder*()

.id(1)

.nom("BAILLE")

.prenom("Anne-Lise")

.email("abaille@campus-eni.fr")

.immatriculation("ENI\_Ecole\_012892")

.build();

employeService.ajouter(employe1);

//Définir le comportement du Repository avec findByImmatriculation

*when*(employeRepository.findByImmatriculation("ENI\_Ecole\_012892")).thenReturn(employe1);

Employe employe2 = Employe

.*builder*()

.id(2)

.nom("GOBIN")

.prenom("Stephane")

.email("sgobin@campus-eni.fr")

.immatriculation("ENI\_Ecole\_012892")

.build();

*assertThrows*(RuntimeException.**class**, () -> employeService.ajouter(employe2));

}

@Test

**void** test\_chargerTousEmployes\_isEmpty() {

**final** List<Employe> lstEmployes = employeService.chargerTousEmployes();

*assertThat*(lstEmployes).isEmpty();

}

@Test

**void** test\_chargerTousEmployes() {

// Arrange

List<Employe> employes = **new** ArrayList<>();

employes.add(

Employe

.*builder*()

.id(1)

.nom("BAILLE")

.prenom("Anne-Lise")

.email("abaille@campus-eni.fr")

.immatriculation("ENI\_Ecole\_012892")

.build());

employes.add(

Employe

.*builder*()

.id(2)

.nom("GOBIN")

.prenom("Stephane")

.email("sgobin@campus-eni.fr")

.immatriculation("ENI\_Ecole\_012111")

.build());

*when*(employeRepository.findAll()).thenReturn(employes);

// Act

List<Employe> result = employeService.chargerTousEmployes();

// Assert

*assertEquals*(2, result.size());

*assertEquals*("ENI\_Ecole\_012892", result.get(0).getImmatriculation());

*assertEquals*("ENI\_Ecole\_012111", result.get(1).getImmatriculation());

}

}

* Nous vérifions au travers de ces tests les comportements de la méthode ajouter de la couche BLL
  + L’ajout d’un employé complet et avec une immatriculation unique se passe bien
  + L’ajout d’un employé nul est impossible
  + L’ajout avec une chaîne de caractère nulle ou vide est impossible
  + L’ajout avec une immatriculation nulle ou vide est impossible
  + L’ajout avec une immatriculation déjà existante est impossible
  + La récupération d’une liste d’employé
* Nous vérifions le comportement de retourner l’ensemble des employés
  + Soit il n’y en a aucun en base et c’est vide
  + Soit il y en a et ils sont retournés