Spring Data Jpa et une transaction

# Démonstration 2 du module 2

|  |
| --- |
| Les objectifs de cette démonstration :   * Déclaration d’une transaction sur la couche BLL * Montrer le tout ou rien |

Déroulement

# Contexte

* Continuer l’application précédente
* Pour le moment nous avons une entité Employe
* Nous voulons ajouter une deuxième entité Adresse
  + Une adresse est composée d’une rue, d’un code postal et d’une ville
  + Pour la base de données nous lui associons un identifiant
* Voici les tables des 2 entités :

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

* Nous voudrions une méthode dans la couche métier, permettant de sauvegarder les 2 entités en une transaction
  + Nous verrons un peu plus tard avec la notion d’association que l’ORM peut le gérer lui même
* Pour montrer l’efficacité d’une transaction, il faut faire échouer une mise à jour (INSERT, UPDATE ou DELETE)
  + Il faut qu’il ait échec en base pour que la transaction se produise
  + Et que l’on obtienne un tout ou rien
  + Nous allons donc peut protéger la méthode métier pour obtenir une erreur

# Création de l’entité : Adresse

* L’entité Adresse doit correspondre à la table : ADDRESS
  + Les annotations nécessaires sont :
    - @ [jakarta](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta).[persistence](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta.persistence).Entity
    - @[jakarta](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta).[persistence](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta.persistence).Table 🡪 pour préciser le nom de la table en base de données
    - @[jakarta](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta).[persistence](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta.persistence).Id 🡪 pour préciser la clef primaire
    - @[jakarta](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta).[persistence](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta.persistence).GeneratedValue 🡪 pour préciser la stratégie de génération. IDENTITY pour notre cas
    - @[jakarta](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta).[persistence](eclipse-javadoc:%E2%98%82=demo-spring-jpa/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/jakarta.persistence%5C/jakarta.persistence-api%5C/3.1.0%5C/66901fa1c373c6aff65c13791cc11da72060a8d6%5C/jakarta.persistence-api-3.1.0.jar=/gradle_used_by_scope=/main,test=/%3Cjakarta.persistence).Column 🡪 pour préciser le nom des colonnes, l’unicité ou non, la nullité ou non et la taille

**package** fr.eni.demo.bo;

**import** jakarta.persistence.\*;

**import** lombok.\*;

@NoArgsConstructor

@AllArgsConstructor

@Data

@Builder

@Entity

@Table(name = "ADDRESS")

**public** **class** Adresse {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

@Column(name = "ADDRESS\_ID")

**private** Integer id;

@Column(name = "STREET", nullable = **false**, length = 250)

**private** String rue;

@Column(name = "POSTAL\_CODE", nullable = **false**, length = 5)

**private** String codePostal;

@Column(name = "CITY", nullable = **false**, length = 150)

**private** String ville;

}

# Création de AdresseRepository

* Créer l’interface AdresseRepository qui hérite de JpaRepository<Adresse, Integer>

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

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

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

**public** **interface** AdresseRepository **extends** JpaRepository<Adresse, Integer>{

}

# Modification de la couche métier (BLL)

Nous allons ajouter une méthode pour créer un nouvel employé et une adresse

* Nous pourrons faire évoluer ce code dans la suite des démonstrations
* Ajouter la méthode void ajouterEmploye(Employe employe, Adresse adresse) dans l’interface EmployeService et son implémentation

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

**import** java.util.List;

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

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

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

**void** ajouter(Employe employe);

List<Employe> chargerTousEmployes();

//Pour valider les transactions

**void** ajouterEmploye(Employe employe, Adresse adresse) ;

}

* + Normalement, dans une couche métier, nous devons créer des validations métiers avant une modification en base
* Dans la classe EmployeServiceImpl, nous nous injectons AdresseRepository
  + Attention, Lombok et l’annotation @AllArgsConstructor 🡪 permet uniquement de créer un constructeur avec tous les attributs
  + Cependant, notre classe de TestEmployeService, n’a que EmployeRepository en paramètre
    - Il va falloir la mettre à jour en mockant AdresseRepository
* Pour montrer l’efficacité d’une transaction, il faut faire échouer une mise à jour
  + Nous allons donc valider les informations de l’employé comme dans la méthode ajouterEmploye précédente
  + Mais aucune validation des données de l’adresse ne sera réalisée
  + Nous allons donc peut protéger la méthode métier pour obtenir une erreur
* C’est la transaction qui devra nous protéger
  + Normalement, nous ferions les 2
  + Vérification métier et transaction
* Nous posons sur la méthode l’annotation @Transactional

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

**import** java.util.List;

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

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

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

**import** jakarta.transaction.Transactional;

**import** lombok.AllArgsConstructor;

//Permet de faire injecter la couche DAL associée

@AllArgsConstructor

@Service

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

**private** EmployeRepository employeRepository;

**private** AdresseRepository adresseRepository;

…

@Override

@Transactional

**public** **void** ajouterEmploye(Employe employe, Adresse adresse) {

// 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.save(employe);

// Validation des données de l'adresse avant sauvegarde

**if** (adresse == **null**) {

**throw** **new** RuntimeException("L'adresse n'est pas renseignée");

}

adresseRepository.save(adresse);

}

}

# Mise à jour de TestEmployeService

* Dans la classe EmployeServiceImpl, nous avons injecté AdresseRepository
  + Attention, Lombok et l’annotation @AllArgsConstructor 🡪 permet uniquement de créer un constructeur avec tous les attributs
  + Cependant, notre classe n’a que EmployeRepository en paramètre
    - Il faut la mettre à jour en mockant AdresseRepository

**…**

**class** TestEmployeService {

**private** EmployeService employeService;

@Mock // Injection d'un Mock du EmployeRepository

**private** EmployeRepository employeRepository;

@Mock // Injection d'un Mock du AdresseRepository

**private** AdresseRepository adresseRepository;

@BeforeEach

**void** init() {

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

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

}

…

}

* Relancer les tests, ils sont tous verts

# Test de transaction

Si nous utilisons des bouchons pour tester les transactions, nous n’aurons pas tout le comportement de la base de données associées

Pour ces tests, nous allons utiliser le contexte de Spring Boot directement

* @SpringBootTest
* Nous serons donc sur la vraie base de données : Microsoft SQL Server

Nous allons utiliser 2 employés et 2 adresses différentes pour bien voir la notion de transaction

* Création d’un test qui permet de valider le comportement dans le cas où tout est bon
  + Avec un employé et la ville de Nantes
* Création d’un second test pour valider le rollback de la transaction complète
  + Avec un second employé et la ville de Rennes

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

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

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

**import** java.util.List;

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

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

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

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

**import** lombok.extern.slf4j.Slf4j;

@Slf4j

@SpringBootTest

**public** **class** TestTransaction {

@Autowired

**private** EmployeService employeService;

@Autowired

**private** EmployeRepository employeRepository;

@Autowired

**private** AdresseRepository adresseRepository;

@Test

**public** **void** test\_transaction\_Ok() {

**final** Adresse adresse = Adresse

.*builder*()

.rue("100 rue de Rennes")

.codePostal("44000")

.ville("Nantes")

.build();

**final** Employe employe = Employe

.*builder*()

.nom("GOBIN")

.prenom("Stéphane")

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

.immatriculation("ENI\_Ecole\_011112")

.numDom("0288XXXXXX")

.build();

// Appel du comportement

employeService.ajouterEmploye(employe, adresse);

// L'employé a été enregistré

*assertThat*(employe.getId()).isGreaterThan(0);

***log***.info(employe.toString());

// L'adresse a été enregistré

*assertThat*(adresse.getId()).isGreaterThan(0);

***log***.info(adresse.toString());

}

@Test

**public** **void** test\_transaction\_Rollback() {

**final** Adresse adresse = Adresse

.*builder*()

.rue("15 rue de Paris")

.ville("Rennes")

.build();

**final** Employe employe = Employe

.*builder*()

.nom("BAILLE")

.prenom("Anne-Lise")

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

.immatriculation("ENI\_ECOLE\_12398")

.numDom("02XXXXXXXX")

.build();

// Appel du comportement

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

// Vérifions les données en base

List<Adresse> adresses = adresseRepository.findAll();

*assertThat*(adresses).isNotNull();

Adresse adresseDB = adresses

.stream()

.filter(item -> item.getVille().equals("Rennes"))

.findAny()

.orElse(**null**);

*assertThat*(adresseDB).isNull();

List<Employe> employes = employeRepository.findAll();

Employe employeDB = employes

.stream()

.filter(item -> item.getImmatriculation()

.equals("ENI\_ECOLE\_12398"))

.findAny()

.orElse(**null**);

*assertThat*(employeDB).isNull();

}

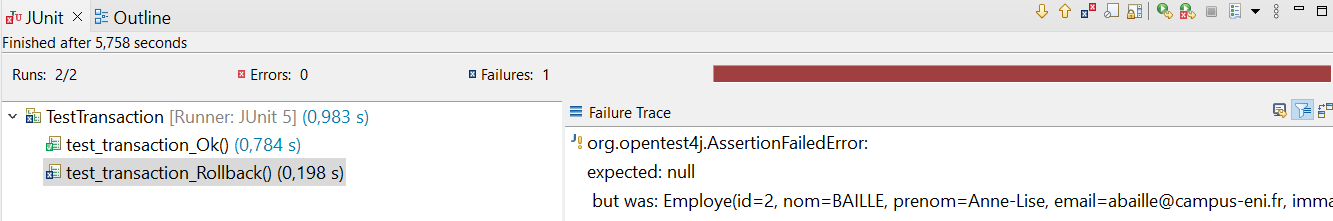
}

## Test avec @Transactional sur la méthode

* Exécuter les tests ; ils sont verts pas de soucis
* Vous pouvez aller voir en base, vous aurez le premier employé et son adresse à Nantes de sauvegardé
  + Le deuxième et son adresse à Rennes n’existe pas
  + Il y a eu un ROLLBACK complet des 2 INSERT (Tout ou rien)

## Test sans @Transactional sur la méthode

* Mettre en commentaire l’annotation
* Relancer les tests :



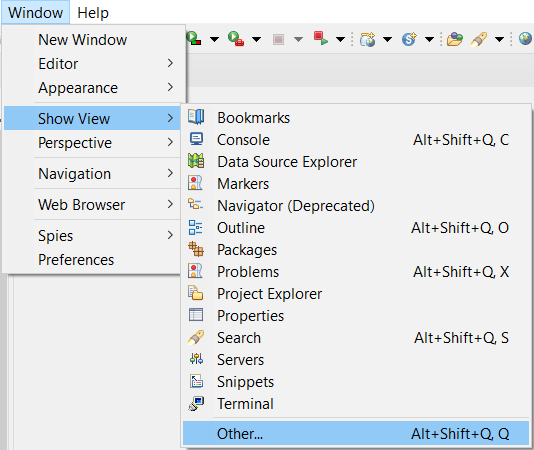
* Celui avec toutes les informations existantes est valide et le premier employé à Nantes est enregistré
* Pour le second test :
  + L’adresse n’a pas été enregistrée car l’attribut codePostal ne peut pas être null
  + Cependant, l’employé a été enregistré. Car la transaction n’englobait pas les 2 INSERT
  + Il y a eu ROLLBACK que de l’insertion de l’adresse pas de l’employé

Il est donc important au niveau BLL, de se demander si les mises à jour doivent être en tout ou rien

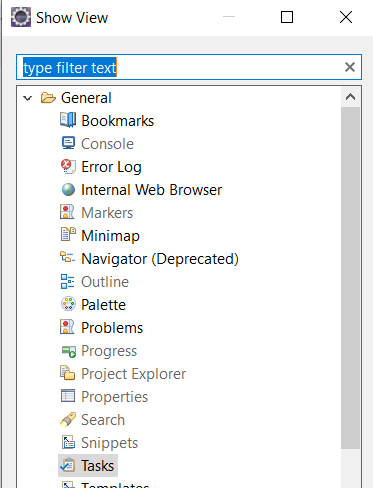
* Dans ce cas, il faut ajouter l’annotation @Transactional sur les méthodes
* Redécommenter l’annotation et relancer les tests pour qu’ils soient verts.

Avantage du TODO

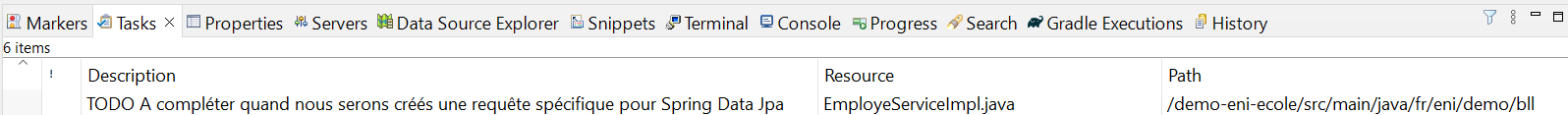
* Nous pouvons les faire afficher en sélectionnant l’onglet : Tasks
* Window 🡪 Show View 🡪 Other …



* General 🡪 Tasks



* Vous avez un nouvel onglet, qui permet de voir et d’accéder aux TODOs



Lors d’un livrable au client, il ne doit plus avoir de TODO

# Création de tests unitaires pour valider le Repository

Spring Boot nous fournit un contexte de tests unitaires pour les JpaRepository.

* Pour cela, il faut utiliser l’annotation :

@org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest

* Et utiliser un « mock » de l’EntityManager de JPA qui est :

org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager

Dans les tests unitaires, nous allons vérifier les comportements de base :

* Création d’un employé
* Trouver un employé par son identifiant
* Mise à jour d’un employé
* Suppression de l’employé
* Récupération de tous les employés

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

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

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

**import** java.util.ArrayList;

**import** java.util.List;

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

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

**import** org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;

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

**import** lombok.extern.slf4j.Slf4j;

@Slf4j

@DataJpaTest

**public** **class** TestEmployeRepository {

@Autowired

**private** TestEntityManager entityManager;

@Autowired

EmployeRepository repository;

@Test

**public** **void** test\_save() {

**final** Employe entiteASauver = Employe

.*builder*()

.nom("BAILLE")

.prenom("Anne-Lise")

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

.immatriculation("ENI\_ECOLE\_12398")

.numDom("02XXXXXXXX")

.build();

// Appel du comportement

**final** Employe entite = repository.save(entiteASauver);

***log***.info(entite.toString());

*assertThat*(entite.getId()).isGreaterThan(0);

}

@Test

**public** **void** test\_findById() {

// Contexte de la DB

**final** Employe entite = Employe

.*builder*()

.nom("DAUTAIS")

.prenom("Servane")

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

.immatriculation("ENI\_ECOLE\_09397")

.numPortable("06XXXXXXXX")

.build();

// Contexte de la DB

entityManager.persist(entite);

entityManager.flush();

***log***.info(entite.toString());

*assertThat*(entite.getId()).isGreaterThan(0);

// Appel du traitement

**int** id = entite.getId();

**final** Optional<Employe> op = repository.findById(id);

//Vérification que l'Optional contienne une entité

*assertThat*(op.isPresent()).isTrue();

//Récupération de l'entité

**final** Employe entiteDB = op.get();

***log***.info(entiteDB.toString());

//Validation de l'entité

*assertThat*(entiteDB.getId()).isEqualTo(id);

*assertThat*(entiteDB).isEqualTo(entite);

}

@Test

**public** **void** test\_update() {

// Contexte de la DB

**final** Employe entite = Employe

.*builder*()

.nom("LACHESNAIS")

.prenom("Frédéric")

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

.immatriculation("ENI\_ECOLE\_15009")

.numDom("02XXXXXXXX")

.numPortable("06XXXXXXXX")

.build();

// Contexte de la DB

entityManager.persist(entite);

entityManager.flush();

*assertThat*(entite.getId()).isGreaterThan(0);

***log***.info("ORIGINE : " + entite.toString());

entite.setNom("DELACHESNAIS");

// Appel du traitement

**final** Employe entiteDB = repository.save(entite);

*assertThat*(entiteDB.getId()).isEqualTo(entite.getId());

*assertThat*(entiteDB.getNom()).isEqualTo(entite.getNom());

***log***.info("MISE A JOUR : " + entiteDB.toString());

}

@Test

**public** **void** test\_delete() {

**final** Employe entite = Employe

.*builder*()

.nom("BAILLE")

.prenom("Anne-Lise")

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

.immatriculation("ENI\_ECOLE\_12398")

.numDom("02XXXXXXXX")

.build();

// Contexte de la DB

entityManager.persist(entite);

entityManager.flush();

***log***.info(entite.toString());

*assertThat*(entite.getId()).isGreaterThan(0);

// Appel du comportement

repository.delete(entite);

// Vérification que l'entité a été supprimée

Employe entityDB = entityManager.getEntityManager().find(Employe.**class**, entite.getId());

*assertNull*(entityDB);

}

// Vérifier que s’il n’y a pas d’enregistrement en base, nous remontons vide

@Test

**public** **void** test\_find\_all\_is\_empty() {

**final** List<Employe> lstEntites = repository.findAll();

*assertThat*(lstEntites).isEmpty();

}

// Retrouver l’ensemble du jeu de données

@Test

**public** **void** test\_findAll() {

// Contexte de la DB

jeuDeDonnees();

**final** List<Employe> lstEntites = repository.findAll();

*assertThat*(lstEntites.size()).isGreaterThan(0);

***log***.info(lstEntites.toString());

}

// Création d’un jeu de données pour simuler la base

**private** **void** jeuDeDonnees() {

**final** List<Employe> lstEntites = **new** ArrayList<>();

lstEntites.add(Employe

.*builder*()

.nom("BAILLE")

.prenom("Anne-Lise")

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

.immatriculation("ENI\_ECOLE\_12398")

.numDom("02XXXXXXXX")

.build());

lstEntites.add(Employe

.*builder*()

.nom("DAUTAIS")

.prenom("Servane")

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

.immatriculation("ENI\_ECOLE\_09397")

.numPortable("06XXXXXXXX")

.build());

lstEntites.add(Employe

.*builder*()

.nom("DELACHESNAIS")

.prenom("Frédéric")

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

.immatriculation("ENI\_ECOLE\_15009")

.numDom("02XXXXXXXX")

.numPortable("06XXXXXXXX")

.build());

// Contexte de la DB

lstEntites.forEach(e -> {

entityManager.persist(e);

});

entityManager.flush();

}

}

Les tests fonctionnent indépendamment les uns des autres.

* Si vous les exécuter, ils seront tous verts.

# Modification des tests unitaires du service

* Dès qu’une mise à jour d’une méthode est faite ; il faut mettre à jour le test unitaire associé
  + Il faut aussi par défaut les faire retourner pour valider les non-régression
* Les tests doivent refléter les mises à jour et continuer à prouver le fonctionnement
* Dans l’itération précédente, la couche d’accès aux données était un bouchon
  + Nous devons donc reprendre les tests unitaires et utiliser Mockito pour gérer le bouchon du EmployeRepository
* Nous aurons pour le moment un test qui sera en échec celui de la recherche d’un employé par son immatriculation
  + Il permettra de confirmer qu’il faut mettre à jour ce code
* Il faut injecter EmployerRepository à la place d’EmployeDAO
* Et changer les méthodes du CRUD d’origine par les méthodes de CrudRepository et JpaRepository

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

**…**

**import** java.util.Optional;

…

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

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

**class** TestEmployeService {

**private** EmployeService employeService;

@Mock // Injection d'un Mock du EmployeRepository

**private** EmployeRepository 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")

.numDom("0299XXXXXX")

.numPortable("0699XXXXXX")

.build();

// Définir le comportement du Repository mocké

*when*(employeRepository.findById(id)).thenReturn(Optional.*of*(employe));

// Comportemnet à valider

employeService.ajouter(employe);

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

Optional<Employe> op = employeRepository.findById(id);

*assertNotNull*(op);

*assertThat*(op.isPresent()).isTrue();

**final** Employe employeDB = op.get();

*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());

}

…

// 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")

.numDom("0299XXXXXX")

.numPortable("0699XXXXXX")

.build();

employeService.ajouter(employe1);

// Définir le comportement du Repository avec findByImmatriculation

//**TODO** A compléter quand nous serons créés une requête spécifique pour Spring Data Jpa

//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")

.numDom("0288XXXXXX")

.build();

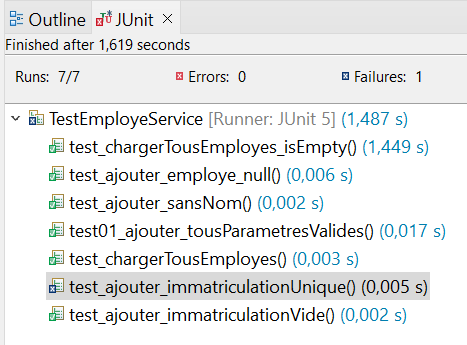
*assertThrows*(RuntimeException.**class**, () -> employeService.ajouter(employe2));

}

…

}

* Lors de l’exécution, les tests passent sauf celui de l’unicité de l’immatriculation



* Dans Tasks nous voyons nos 2 TODOs

